

SEHH2239 Data Structures

Lecture 11

Learning Objectives:

- To describe the purpose of hashing
- To explain the collisions in hashing
- To solve collisions with linear probing and separate chaining

Introduction

- Consider the problem of **searching** an array for a given value
 - If the array is not sorted,
 - If the value isn't there, we need to search all n elements
 - If the value is there, we search $n/2$ elements on average
 - If the array is sorted, we can do a binary search
 - About equally fast whether the element is found or not
 - It doesn't seem like we could do much better
 - That is **constant time search?**
 - We can do it if the array is organized in a particular way

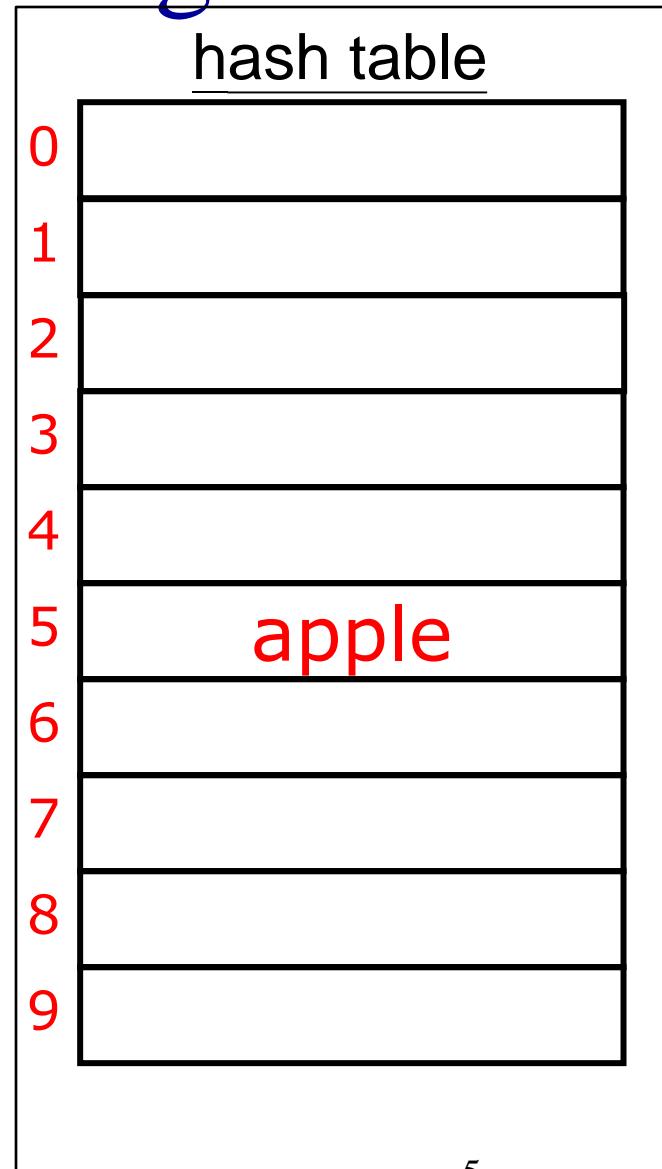
Hashing

- Recall the {key, value} pairs.
- Questions:
 - How to store the keys?
- **Hashing** is a technique that determines the *index* of a *key*.
 - Not search for the item
- **Hash Table**
 - Array storing values indexed by a hash function

Example hashing

- Suppose our hash function gave us the following values:

`hashCode("apple") = 5`



Example hashing

- Suppose our hash function gave us the following values:

`hashCode("apple") = 5`

`hashCode("watermelon") = 3`

`hashCode("grapes") = 8`



Example hashing

- Suppose our hash function gave us the following values:

hashCode("apple") = 5

hashCode("watermelon") = 3

hashCode("grapes") = 8

hashCode("cantaloupe") = 7

hashCode("kiwi") = 0

hashCode("strawberry") = 9

hashCode("mango") = 6

hashCode("banana") = 2

hash table	
0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Getting value from key

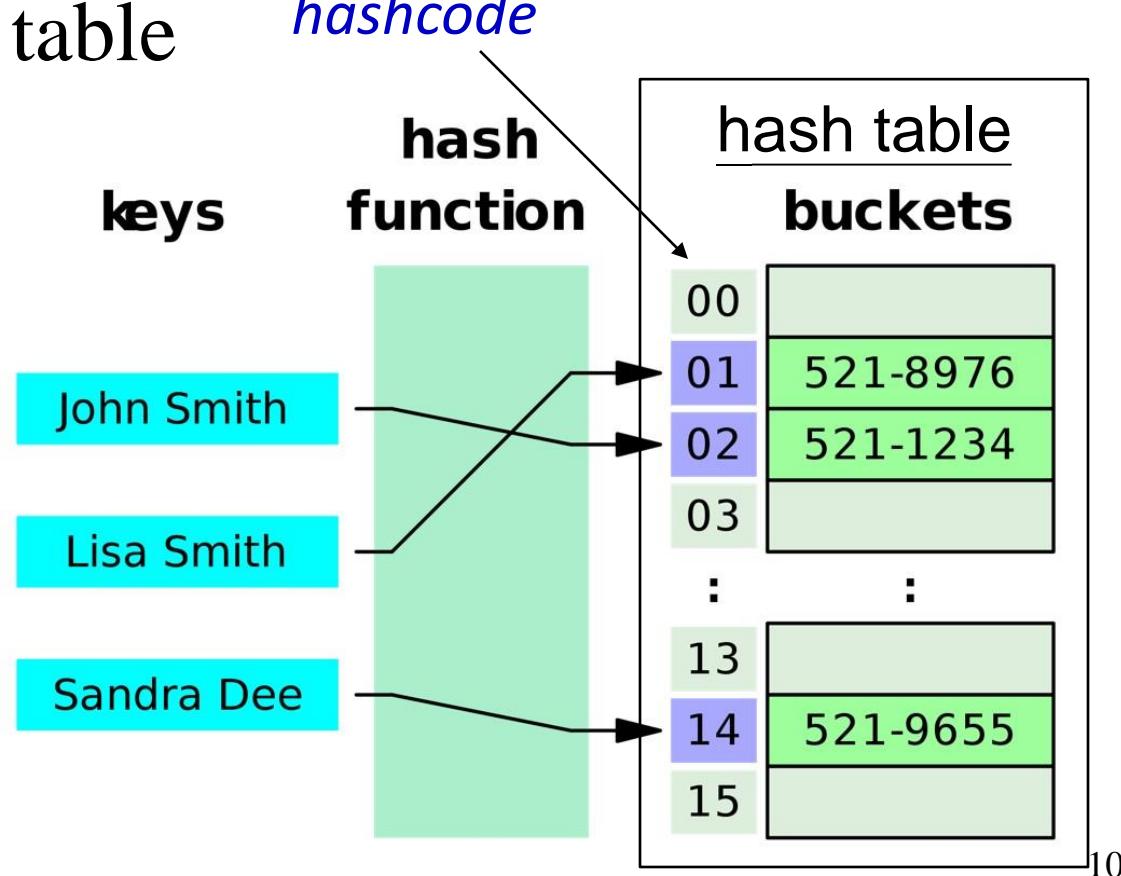
- Storing {key, value} pair
 - We use a *key* to find a place in the hash table
 - The associated *value* is the information we are trying to look up
- Example:
 - $\text{hash}(\text{robin}) = 142$
 - $\text{hash}(\text{owl}) = 148$

	<i>key</i>	<i>value</i>
...		
141		
142	robin	robin info
143	sparrow	sparrow info
144	hawk	hawk info
145	seagull	seagull info
146		
147	bluejay	bluejay info
148	owl	owl info

Hash function

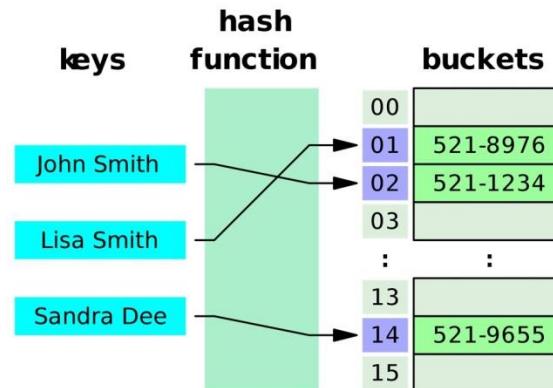
Hash function

- A hash function takes a search *key* and produces the *integer index* of an element in the hash table



Use of Hash function

- Uses a 1D array (or table) $\text{table}[0:b-1]$.
 - Each position of this array is a **bucket**.
 - A bucket can normally hold only one dictionary pair.
- Uses a hash function f that converts each key k into an index in the range $[0, b-1]$.
 - $f(k)$ is the **home bucket** for key k .
- Every dictionary pair (**key, element**) is stored in its home bucket $\text{table}[f(\text{key})]$.



Looking up with Hash function

- Given a key to look up for, it would tell us exactly where in the array to look
 - If it's in that location, it's in the array
 - If it's not in that location, it's not in the array

Hash function

- A **hash function** is a function that:
 - When applied to an Object, returns a number (index)
 - When applied to *equal* Objects, returns the *same* number for each
 - When applied to *unequal* Objects, is *very unlikely* to return the same number for each
- Preliminary examples of hash functions:
 - $\text{hash}(X) = X/n$ where n is 11
 - $\text{hash}(X) = X \% \text{ noOfBuckets}$

Example of Hashing

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is `table[0:7]`, $b = 8$.
- Hash function is $h(key) = key/11$.
- Pairs are stored in table as below:

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Perfect Hashing and Collision

Prefect Hashing

- A Perfect hashing maps each *search key* into a *different integer* that is suitable as an index to hash table
- Efficiency
 - Can result in $O(1)$ search times

Examples of Perfect Hashing

hash table	
0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

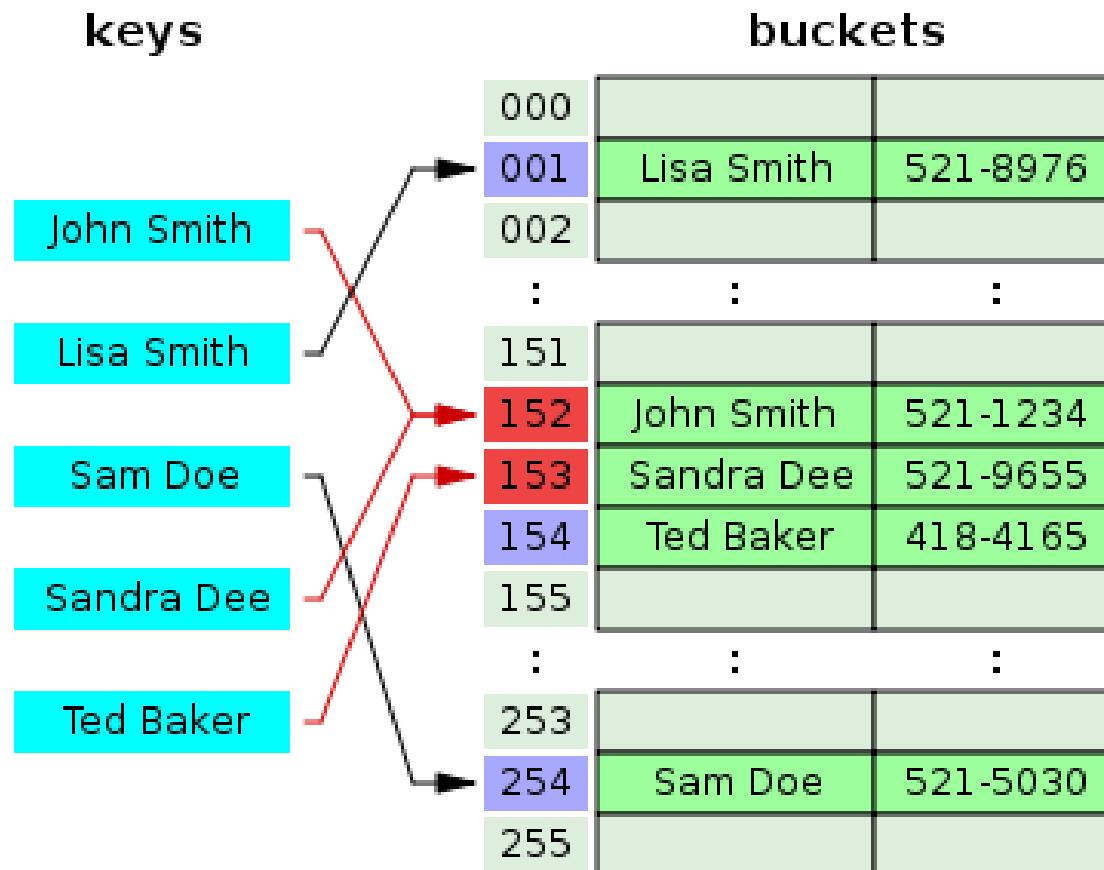
- Where does $(26,g)$ go?
- Keys that have the same home bucket are **synonyms**.
 - 22 and 26 are synonyms with respect to the hash function that is in use.
- The home bucket for $(26,g)$ is already occupied.



Collisions

- When two values hash to the same array location, this is called a **collision**
- Collisions are normally treated as “**first come, first served**”—the first value that hashes to the location gets it
- We have to find something to do with the second and subsequent values that hash to this same location

Collision



Handling collisions

- What can we do when two different values attempt to occupy the same place in an array?
 - **Solution #1:** Linear probing - search from there for an **empty location**
 - Can stop searching when we find the value *or* an empty location
 - Search must be end-around
 - **Solution #2:** Use the array location as the header of a **linked list** of values that hash to this location
- All these solutions work, provided:
 - We use the same technique to *add* things to the array as we use to *search* for things in the array

Collision Handling by Linear Probing

Linear probing

- all entry records are stored in the array itself.
- When a new entry has to be inserted, compute the hashed value
 - If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index (as the Perfect hashing).
 - If the slot is occupied, it proceeds in some probe sequence until it finds an unoccupied slot.

Insertion - Case 1

- Suppose you want to add **seagull** to this hash table
- Also suppose:
 - `hashCode(seagull) = 143`
 - `table[143]` is not empty
 - `table[143] != seagull`
 - `table[144]` is not empty
 - `table[144] != seagull`
 - `table[145]` is empty
- Therefore, put **seagull** at location 145

...	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
...	

Searching - Case 1

- Suppose you want to look up **seagull** in this hash table
- Also suppose:
 - `hashCode(seagull) = 143`
 - `table[143]` is not empty
 - `table[143] != seagull`
 - `table[144]` is not empty
 - `table[144] != seagull`
 - `table[145]` is not empty
 - `table[145] == seagull !`
- We found **seagull** at location 145

...	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
...	

Searching - Case 2

- Suppose you want to look up **cow** in this hash table
- Also suppose:
 - `hashCode(cow) = 144`
 - `table[144]` is not empty
 - `table[144] != cow`
 - `table[145]` is not empty
 - `table[145] != cow`
 - `table[146]` is empty
- If **cow** were in the table, we should have found it by now
- Therefore, it isn't here

...	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
...	

Case 3

- Suppose you want to add **hawk** to this hash table
- Also suppose
 - `hashCode(hawk) = 143`
 - `table[143]` is not empty
 - `table[143] != hawk`
 - `table[144]` is not empty
 - `table[144] == hawk`
- **hawk** is already in the table, so do nothing in adding.

...	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
...	

Case 4

- Suppose:
 - You want to add **cardinal** to this hash table
 - `hashCode(cardinal) = 147`
 - The last location is 148
 - 147 and 148 are occupied
- Solution:
 - Treat the table as **circular**; after 148 comes 0
 - Hence, **cardinal** goes in location 0 (or 1, or 2, or ...)

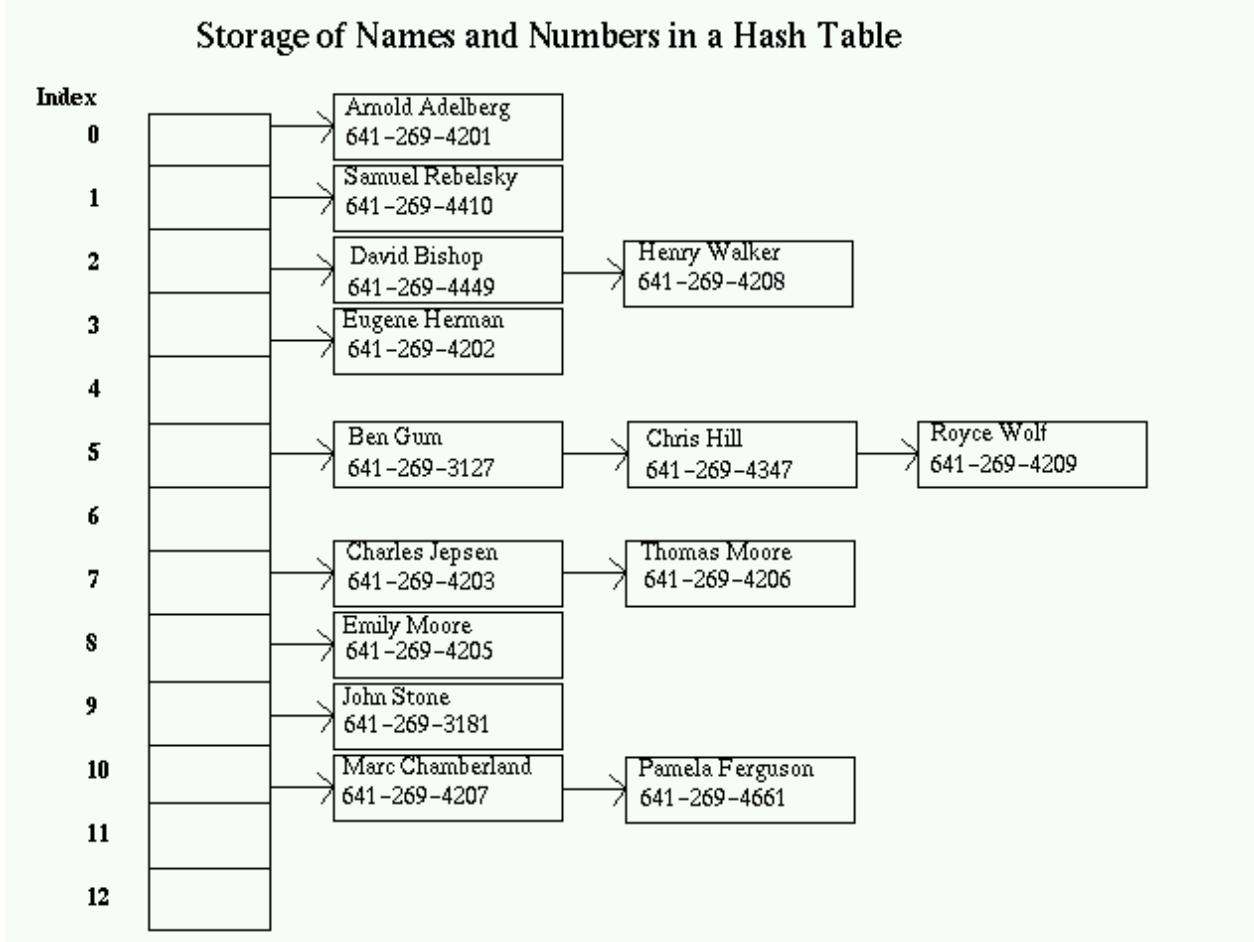
...	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl

Collision Handling by Separate chaining

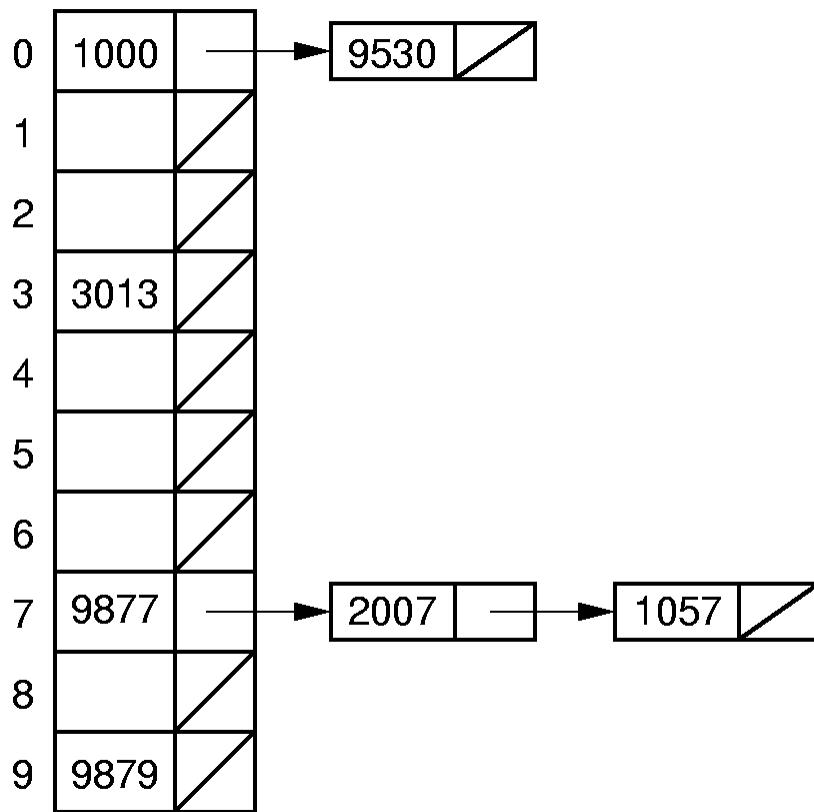
Separate chaining

- In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list.
- If there is any collision, store both the elements in the same linked list.

Separate chaining – I



Separate chaining - II



Overflow

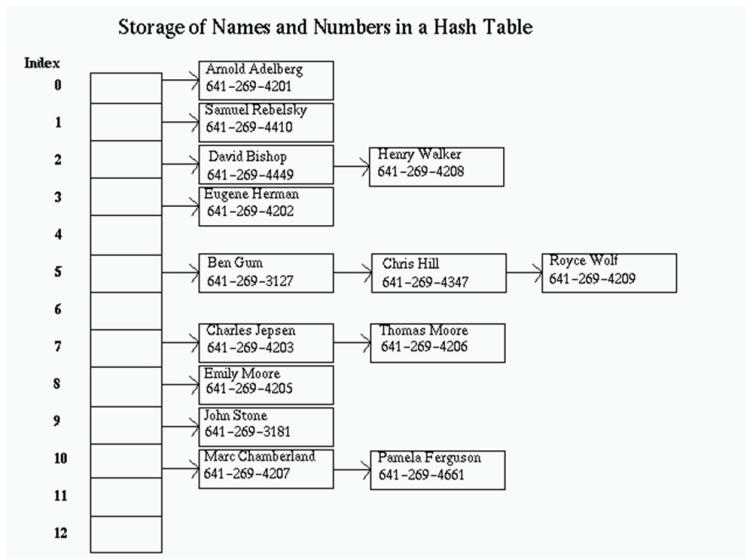
Overflow

(3,d)	(11,m)	(22,a)	(33,c)	(45,k)	(55,o)	(73,e)	(85,f)
-------	--------	--------	--------	--------	--------	--------	--------

- An overflow occurs when there is no space in the home bucket for the new pair.
- When a bucket can hold only one pair, collisions and overflows occur together.

Handling Overflow

- Linked list can also resolve the problem, as linked list can be expanded to increase the capacity.



Hashing in Python and Efficiency

Hash table

- In Python, the Dictionary data types represent the implementation of hash tables. The Keys in the dictionary satisfy the following requirements.
 - The keys of the dictionary are hashable i.e. they are generated by hashing function which generates unique result for each unique value supplied to the hash function.
 - The order of data elements in a dictionary is not fixed.

Example

```
# Declare a dictionary
dict = { 'Name': 'Zara', 'Age': 7, 'Class': 'First' }

# Accessing the dictionary with its key
print("dict['Name']: ", dict['Name'])
print("dict['Age']: ", dict['Age'])
```

Output:

```
dict['Name']: Zara dict['Age']: 7
```

Prefect Hashing

- Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time.

Good Hash Functions

- General characteristics of hash functions $h(key)$.
Any function can be a hash function that **distributes entries uniformly** throughout the hash table.
- Good hash functions:
 - Minimize collision
 - Be fast to compute
- There are many ways to implement hash functions, e.g. to use *prime number division*, *mid square*, *move or folding* just to mention a few, but they are beyond the scope of the class.

Efficiency

- Hash tables are actually surprisingly efficient
- Until the table is about 70% full, the number of **probes** (places looked at in the table) is typically only 2 or 3
- Sophisticated mathematical analysis is required to *prove* that the expected cost of inserting into a hash table, or looking something up in the hash table, is constant time
- Even if the table is nearly full (leading to occasional long searches), efficiency is usually still quite high

Summary of key terms

- Hash
 - hashing
 - Hash function
 - Perfect Hashing
- Collision
 - Linear probing
 - Separate chaining
- Overflow

LECTURE I: DATA STRUCTURE AND INTRODUCTION TO PYTHON

SEHH2239 Data Structures

LEARNING OBJECTIVES:

- To overview the significance of data structure
- To review the elements in a python program
- To review the expressions, operators, and flow control in python programs
- To declare list structure and access list elements

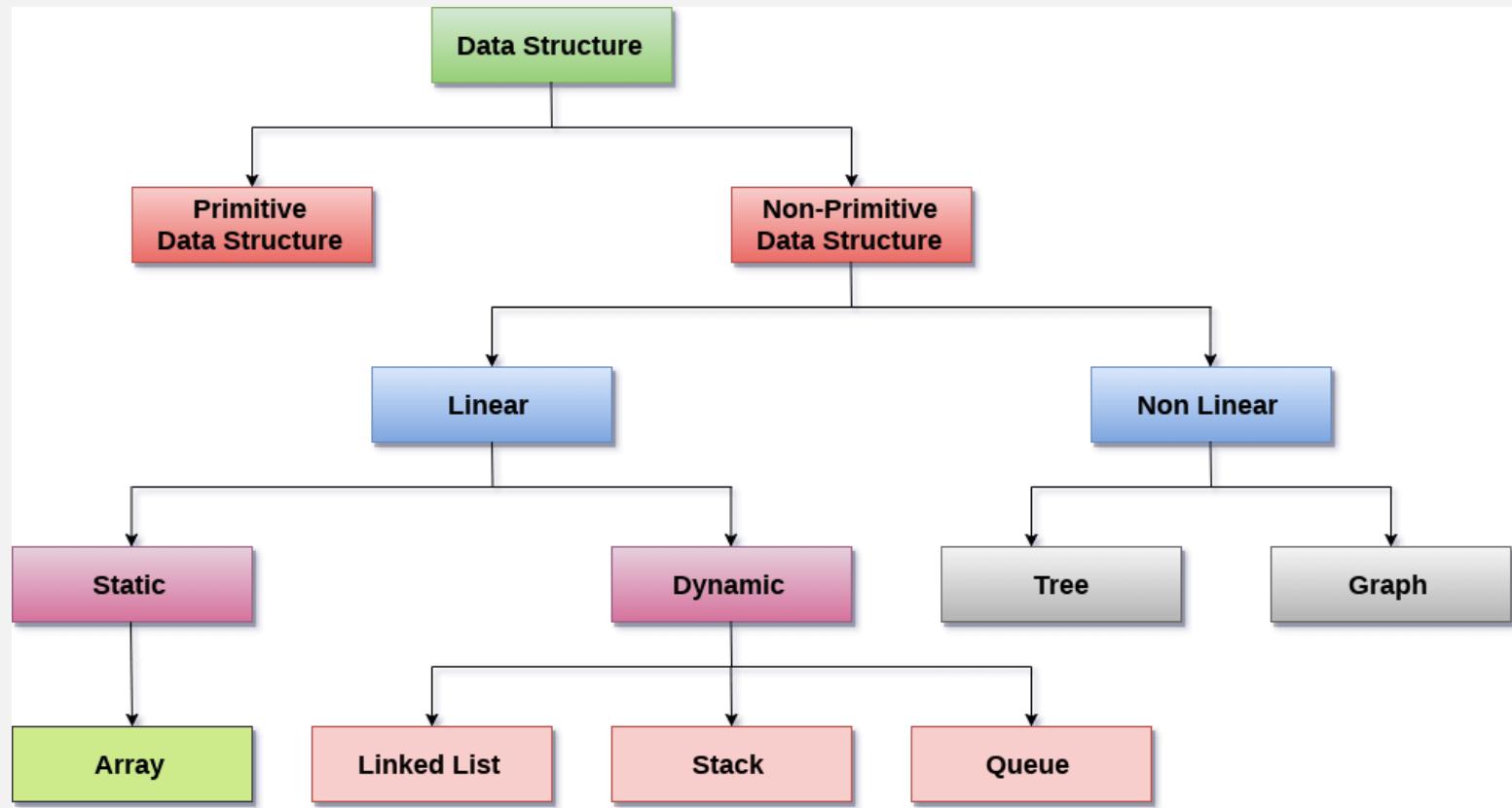
DATA STRUCTURE

- **Data**
 - can be defined as a representation of facts, concepts, or instructions in a formalized manner.
 - Data is represented with the help of characters such as alphabets (A-Z, a-z), digits (0-9) or special characters (+,-,/,*,<,>,= etc.)
- A **data structure** is a specialized format for
 - organizing data,
 - processing data,
 - retrieving data and
 - storing data.
- In computer programming, a data structure may be selected or designed to **store data** for the purpose of **working on it** with various **algorithms**.
- Each data structure contains information about the data *values, relationships between the data and functions* that can be applied to the data.

IMPORTANCE OF DATA STRUCTURES

- Data structures are essential for **managing large amounts of data**, such as information kept in databases or indexing services, efficiently.
- Proper maintenance of data systems requires the **identification of memory allocation, data interrelationships** and **data processes**, all of which data structures help with.
- Choosing an ill-suited data structure could result **in slow runtimes or unresponsive code**.
- A few factors to consider when picking a data structure include
 - what **kind** of information will be stored,
 - **where** should existing data be placed,
 - how should data be **sorted** and
 - **how much memory** should be reserved for the data

DATA STRUCTURE CLASSIFICATION



<https://bcastudyguide.wordpress.com/unit-1-introduction-to-data-structure-and-its-characteristics/>

OVERVIEW OF TYPES OF DATA STRUCTURES

Primitive and non-primitive

- **Primitive** are predefined types of data supported by the programming language.
 - For example, integers, float numbers, and characters are all primitive data types. E.g. 3, 0.45, s, ^...
- **Non-primitive** are not defined by the programming language but are instead created by the programmer.
 - They are sometimes called "reference variables" or "object references" since they reference a memory location, which stores the data.

INTRODUCTION TO PYTHON

WHAT IS PYTHON?

- Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.
- It is used for:
 - web development (server-side),
 - software development,
 - mathematics,
 - system scripting.
- What can Python do?
 - Python can be used on a server to create web applications.
 - Python can connect to database systems. It can also read and modify files.
 - Python can be used to handle big data and perform complex mathematics.
 - Python can be used for rapid prototyping, or for production-ready software development.

HELLO WORLD IN PYTHON

- In Google colab
 - <https://colab.research.google.com/notebooks/>

```
print("Hello World!")
```

- In IDE (e.g. PyCharm)
 - Save it as `helloWorld.py`

PYTHON EXPRESSIONS AND OPERATORS

Hello my name is John
Hello

COMPONENTS OF A PYTHON PROGRAM

- Executable statements are placed in **functions**
- Known as **methods**, that belong to class definitions.



```
class Person:  
    def __init__(self, name):  
        self.name = name  
  
    def myfunc(self):  
        print("Hello my name is " + self.name)  
  
p1 = Person("John")  
p1.myfunc()
```

>>Hello my name is John

EXPRESSIONS AND OPERATORS

- The semantics of an **operator** depends upon the type of its **operands**.
 - For example
 1. a and b are **numbers**, the syntax $a + b$ indicates **addition**
 2. a and b are **strings**, the *operator* `+` indicates **concatenation**
 - **Assignment Operator (=)**
 - = is used in python to assign values to variables

NUMBERS IN PYTHON

- Different numeric types in Python:

- **Int**

- $x = 5$

- **Float**

- $y = 3.46$

- **Complex**

- $z = 2 + 5j$



To view the number type:

```
print(type(x))  
>><class 'int'>
```

```
print(type(y))  
>><class 'float'>
```

```
print(type(z))  
>><class 'complex'>
```

- Scientific numbers

- $m = 2e5$

- $n = 3.53E4$



To print the number:

```
print(m)  
>> 200000.0
```

```
print(n)  
>> 35300.0
```

STRINGS

- Both `""` or `'` can be used to quote strings
 - `a="ABC"`
 - `b1='dddggg lll llll .adf'`
 - `c_dfasdf_= '123123'`
 - `d="""swing swimming so many medals add oil!"""`

To print the string

```
print(a[2])  
>> C
```

```
print(d)  
>>'swing swimming so many medals add oil!'
```

STRINGS

- Get a character from a string

```
print(a[2])  
>>C
```

- String length

```
print(len(a))  
>>3  
print(len(b1))  
>>20
```

- Slicing

```
print(b1[3:8])  
>>ggg 1
```

- Search in a string

```
print("silver" in d)  
>>False
```

```
if "medals" in d:  
    print("yes, found")  
>>yes, found
```

ESCAPE CHARACTERS

- Allows you to use some special characters

```
print("\t\"hello world\")
```

```
>>     "hello world"
```

Escape Sequence	Meaning
\newline	Backslash and newline ignored
\\\	Backslash (\)
\'	Single quote ('')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value <i>ooo</i>
\xhh	Character with hex value <i>hh</i>

CASTING OF NUMBERS

- Casting in python is done by **constructor** functions:
- **int()** - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
- **float()** - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- **str()** - constructs a string from a wide variety of data types, including strings, integer literals and float literals

```
x = 5.5  
y = int(x)  
print(y)
```

ARITHMETIC OPERATORS

- Python supports the following arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
%	the modulo operator

- If both operands have type int, then the result is an int
- If one or both operands have type float, the result is a float.
- Integer division has its result truncated.

Try this: operators.ipynb

https://www.w3schools.com/python/python_operators.asp

INCREMENT AND DECREMENT OPS

- Python does not have unary increment/decrement operator (++/--). Instead to increment a value, use

```
a=2  
a += 1  
print(a)
```

```
>>3
```

LOGICAL OPERATORS

- Python supports the following operators for numerical values, which result in *Boolean values*:

- `x < y`
- `S == "abc"`
- `b1 != true`

<code><</code>	less than
<code><=</code>	less than or equal to
<code>==</code>	equal to
<code>!=</code>	not equal to
<code>>=</code>	greater than or equal to
<code>></code>	greater than

- Boolean values also have the following operators:

- `x < y && S == "abc"`
- `!(x < y)`
- `(S == "abc" || b1 != true)`

<code>!</code>	not (prefix)
<code>&&</code>	conditional and
<code> </code>	conditional or

PYTHON INDENTATION

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

```
y = 14
if y<5:
    print("this line run when true")
print("this line run anyway")
```

```
>>this line run anyway
```

FLOW CONTROL

IF STATEMENTS

- The syntax of a simple **if** statement is as follows:

```
x=5  
y=13  
if y>x:  
    print("y is greater.")
```

>>y is greater.

- if...else...**

```
y=2  
if y>x:  
    print("y is greater.")  
else:  
    print("y is not greater.")
```

>>y is not greater.

COMPOUND IF

- There is also a way to group a number of boolean tests, as follows:

```
if firstBooleanExp:  
    firstBody  
elif secondBooleanExp:  
    secondBody  
else:  
    thirdBody
```

```
if y>x:  
    print("y is greater.")  
elif y==x:  
    print("they are equal.")  
else:  
    print("y is smaller.")
```

- Short-hand if...else (Ternary Operator)

```
variable = expressionTrue if condition else expressionFalse;
```

```
# Python Ternary Operator  
ylarge = True if y >x else False
```

WHILE LOOPS

- Such a loop tests that a certain condition is satisfied and will perform the body of the loop each time this condition is evaluated to be true.
- The syntax for such a conditional test before a loop body is executed is as follows:

```
while booleanExpression:  
    loopBody
```

```
i=1  
while i<6:  
    print(i)  
    i+=1
```

```
>>  
1  
2  
3  
4  
5
```

BREAK AND CONTINUE

- **break** statement that immediately terminate a while or for loop when executed within its body.

```
a = 3
while a < 7:                                >>
    print(a)                                 3
    if a == 4:
        break
    a += 1                                 4
```

- **continue** statement that causes the current iteration of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

```
a = 3
while a < 7:                                >>
    a +=1                                 4
    if a == 5:
        continue
    print(a)                             6
                                         7
```

FOR LOOPS

- The traditional **for-loop** syntax consists of four sections:
 - an initialization
 - a boolean condition
 - an increment statement
 - and the body—although any of those can be empty.
- The basic structure is as follows:

```
for counter in range(m, n)
    loopBody
```

```
for i in range(0, 4):
    print(i)
```

```
>>
0
1
2
3
```

VARIABLE NAMES IN PYTHON

- Rules for Python variables:
 - A variable name must start with a letter or the underscore character
 - A variable name cannot start with a number
 - A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
 - Variable names are case-sensitive (age, Age and AGE are three different variables)

KEYWORDS IN PYTHON

- Keywords in Python are reserved words that cannot be used as a variable name.

```
help("keywords")
```

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

SUMMARY OF KEY TERMS

- Data Structure
- Basic Python
 - Expressions and Operators
 - Arithmetic
 - Logical
 - Relational
 - Flow Control
 - If
 - while/do while / for

REFERENCES:

- Textbook:
 - **Adam Drozdek, Data Structures and Algorithms in Python , 1st Edition, Cengage Learning, 2021**
- References
 - https://www.tutorialspoint.com/computer_fundamentals/computer_data.htm
 - <https://searchsqlserver.techtarget.com/definition/data-structure>
 - <https://www.w3schools.com/python/>
 - <https://www.geeksforgeeks.org/array-copying-in-python/>

LECTURE 2

PYTHON PROGRAMMING

SEHH2239 DATA STRUCTURES

LEARNING OBJECTIVES:

- To use Python Functions to structure codes
- To use built-in data structures – Tuple
- Array
- To define classes to represent objects
- To work out tasks using instance methods

FUNCTIONS

FUNCTIONS

- A function is a block of code which only runs when it is called.
 - You can pass data, known as parameters, into a function.
 - A function can return data as a result.
-
- Note: pass Statements
 - The pass statement does nothing. It can be used when a statement is required syntactically but the program or function requires no action.

DEFINING A FUNCTION AND CALLING

- Defining a function
 - The keyword def introduces a function definition.
 - It must be followed by the function name and the parenthesized list of formal parameters.
 - The statements that form the body of the function start at the next line, and must be indented.

```
def firstFcn():
    print("Printing from a function.")
```

```
def secondFcn(name):
    print("You are " + name)
```

- Return a value

```
def thirdFcn(n, m):
    p = n+ 2*m
    return p
```

CALLING A FUNCTION

- To call a function, use the function name followed by parenthesis.

```
firstFcn()
```

```
secondFcn("Patrick")  
secondFcn("Joseph")
```

- Arguments can be specified after the function name, inside the parentheses. A number of arguments can be added and separate them with commas.
- Calling to get a return value

```
a = thirdFcn(1,2)  
print(thirdFcn(2,3))
```

ARGUMENTS IN FUNCTIONS

- Number of Arguments
 - By default, a function must be called with the correct number of arguments.
- Pass by reference vs value
 - All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.
- Default arguments
 - A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

```
def defaultArgFcn(name, area = "HK"):  
    print("You are " + name + " in " + area)
```

```
defaultArgFcn("Joseph")  
defaultArgFcn("Mary", "Chicago")
```

You are Joseph in HK

You are Mary in Chicago

TYPES OF ARGUMENTS

- There are two types of arguments.

```
def defaultArgFcn(name, area):  
    print("Hello " + name + ", you live in " + area)
```

Positional arguments

- Positional arguments are values that are passed into a function based on the order in which the parameters were listed during the function definition.

```
defaultArgFcn("John", "Hong Kong")  
>> Hello John, you live in Hong Kong
```

Keyword arguments

- Keyword arguments (or named arguments) are values that, when passed into a function, are identifiable by specific parameter names. A **keyword** argument is preceded by a parameter and the assignment operator, = .

```
defaultArgFcn(name ="John", area ="Hong Kong")  
>> Hello John, you live in Hong Kong  
defaultArgFcn(area ="KLN", name = "Paul")  
>> Hello Paul, you live in KLN
```

IMPORT STATEMENT

- You can use any Python source file as a module by executing an import statement in some other Python source file.

```
import module1[, module2[, ... moduleN]]
```

- For example, Python has a built-in module that you can use for mathematical tasks.

```
import math
print (math.sqrt (9))      3.0
print (math.sqrt (78))     8.831760866327848
print (math.isqrt (68))    8
```

TUPLE

TUPLE

- A tuple is a collection of objects which ordered and immutable.
- Tuples are **sequences**, just like lists.
- The differences between tuples and lists:
 - Tuples **cannot be changed** (immutable) unlike lists
 - Tuples use parentheses (), whereas lists use square brackets [].
- Creating a tuple is as simple as putting different comma-separated values.

```
t1 = ("apple", "box", "cake", "disk")
print(t1)
>>('apple', 'box', 'cake', 'disk')
t2 = (1, 2, 3, 4, 5 )
t3 = "a", "b", 6, 0x23
print(t3)
>>('a', 'b', 6, 35)
t4 = ()
```

tuple.py

ACCESS ITEMS AND REMOVING TUPLES

- Access items
 - Tuple items by referring to the index number, inside square brackets:

```
print(t1[2])  
>> cake  
print(t2[1:3])  
>>(2, 3)
```

- Updating Tuples or deleting an item
 - Tuples are immutable which means you cannot update or change the values of tuple elements.
 - Removing individual tuple elements is not possible.
- Remove an entire tuple

```
del t2
```

BASIC TUPLES OPERATIONS

```
t1 = ("apple", "box", "cake", "disk")
```

Python Expression	Results	Description
<code>len(t1)</code>	4	Length
<code>t1 + ("egg", "fish")</code>	('apple', 'box', 'cake', 'disk', 'egg', 'fish')	Concatenation
<code>("go!") * 3</code>	('go!', 'go!', 'go!')	Repetition
<code>"box" in t1</code> <code>"ox" in t1</code>	True False	Membership
<code>for x in (2, 3, 6):</code> <code> print(x)</code>	2 3 6	Iteration

INDEXING, SLICING, MATRIXES, MIN AND MAX

```
t1 = ("apple", "box", "cake", "disk")
```

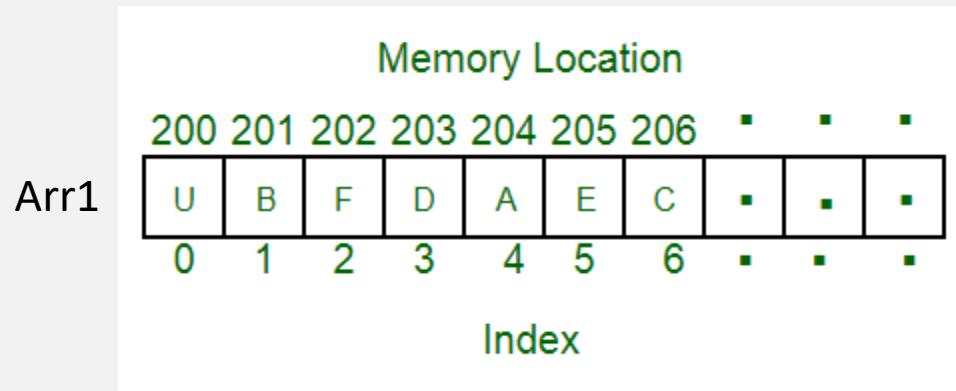
```
t2 = (1, 2, 3, 4, 5)
```

Python Expression	Results	Description
t1[3]	'disk'	Index start at zero
t1[-2]	'cake'	Negative: count from the right
t1[2:]	('cake', 'disk')	Slicing fetches sections
t1[::-2]	('apple', 'cake')	Slicing
t1[::-1]	('disk', 'cake', 'box', 'apple')	Listing from the back
max(t2)	5	Max
min(t2)	1	Min

ARRAYS

ARRAY DEFINITION

- An **array** is a sequenced collection of variables all of the same type.
- Each variable, or **cell**, in an array has an **index**, which uniquely refers to the value stored in that cell. The cells of an array, **Arr1**, are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called an **element** of that array.



- Array is the simplest data structure where each data element can be randomly accessed by using its **index** number.

ARRAYS IN PYTHON

- Python does not have built-in support for Arrays, but **Lists** can be used instead.
- To declare a variable that holds an array of strings as a **list**.
- To insert values to it, we can use an array literal - place the values in a comma-separated list, inside square brackets:

```
sports = ["soccer", "basketball", "badminton", "squash"]
```

- To create an array of integers, you could write:

```
n = [ 10, 20, 35, 40]
```

ACCESS THE ELEMENTS OF A LIST

- You access a list element by referring to the index number.
- This statement accesses the value of the first element in sports:

```
print(sports[2])
```

>>badminton

CHANGE AN ELEMENT IN A LIST

- To change the value of a specific element, refer to the index number:
- Example

```
sports[2] = "tennis"  
print(sports[2])
```

```
>> tennis
```

- Use negative index to access elements from the end.

```
sports[-1]
```

```
>> squash
```

```
sports[-3]
```

```
>>basketball
```

LENGTH OF A LIST

- To find out how many elements a list has, use the *len* property:
- Example:

```
len(sports)
```

```
>> 4
```

LIST SIZE

- Generally, a list is of fixed size and each element is accessed using the indices.
 - For example, we have created a list with size 4.
 - Then the valid expressions to access the elements of this list will be `sports[0]` to `sports[3]` (`length-1`).
 - Whenever you used the value greater than or equal to the size of the list (e.g. `sports[9]` or `sports[20]`), it gives
`IndexError: list index out of range.`
- Caution: The following is not correct.

```
sports[6] ## It will give an error
>>>> IndexError: list index out of range
```

LOOP THROUGH A LIST

- To show the all content in a list, just print the variable name.

```
print(sports)
```

```
>> ['soccer', 'basketball', 'badminton', 'squash']
```

- The following example outputs all elements in the **sports** list with for loop:
- The length property to specify how many times the loop should run.
- Example

```
for a in sports:  
    print(a)
```

```
>>  
soccer  
basketball  
tennis  
Squash
```

LIST OPERATIONS

- Add a new element

```
sports.append("swimming")  
sports
```

```
>>['soccer', 'basketball', 'badminton', 'squash', 'swimming']
```

- Return and Remove an element at the specified position

```
#Remove an element  
sports.pop(2)
```

```
>>badminton
```

```
sports
```

```
>>['soccer', 'basketball', 'squash', 'swimming']
```

LIST OPERATIONS

Remove the first occurrence of the element with the specified value

```
sports.remove("squash")  
sports
```

```
>> ['soccer', 'basketball', 'swimming']
```

Return the position at the first occurrence of the specified value.

```
sports.index("swimming")
```

```
>> 2
```

Inserts the specified value at the specified position.

```
sports.insert(2, "hockey")  
sports
```

```
>> ['soccer', 'basketball', 'hockey', 'swimming']
```

LIST COPYING

Two ways to copy lists :

1. Simply using the assignment operator.
2. Deep Copy

I. Simply using the assignment operator.

Assignment statements do not copy objects, they create bindings between a target and an object.

```
sports = ["soccer", "basketball", "badminton", "squash"]
new_sports = sports
print(id(sports))
>> 140211906678128
print(id(new_sports))
>> 140211906678128
sports[1] = "polo"
print(new_sports[1])
>>polo
```

LIST COPYING

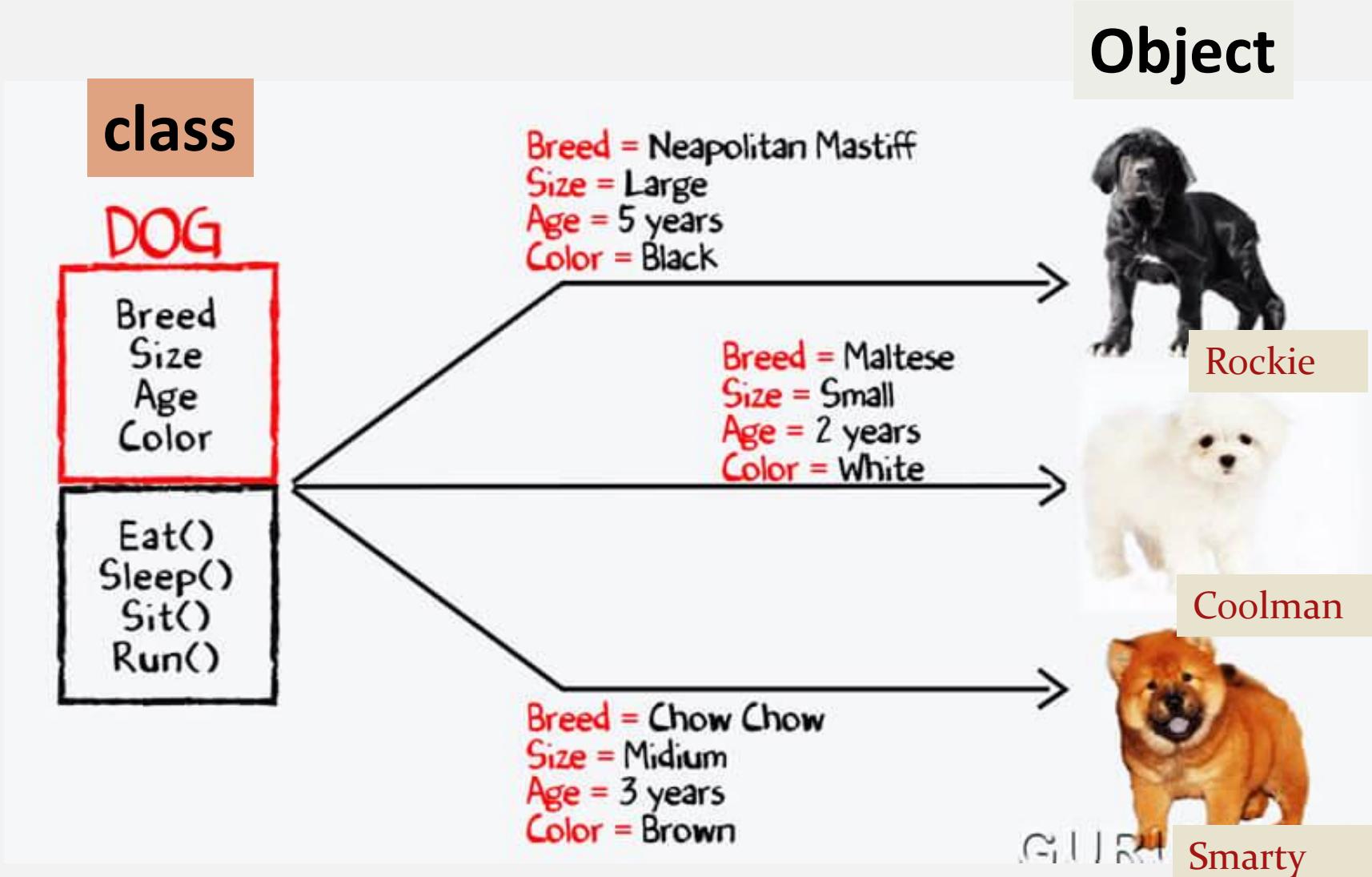
2. Deep Copy

A copy of the object is copied into another object. It means that any changes made to a copy of the object do not reflect in the original object.

```
sports = ["soccer", "basketball", "badminton", "squash"]  
new_sports = sports.copy()  
print(id(sports))  
>> 140211906678128  
print(id(new_sports))  
>> 140211801453104  
sports[1] = "polo"  
print(new_sports[1])  
>>basketball
```

FUNDAMENTALS OF OBJECT- ORIENTED PROGRAMMING IN PYTHON

CLASS AND OBJECTS



CLASS DEFINITIONS

- A **class** serves as the primary means for abstraction in object-oriented programming.
- A **class** provides a set of *behaviors* in the form of member functions (also known as **methods**), with implementations that belong to all its instances.
- A **class** also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, **object variables** or **data members**).

CLASS AND OBJECTS

- Each **object** created in a program is an **instance** of a **class**.
- Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition typically specifies **instance variables**, also known as **data members**, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute.

DEFINE A CLASS IN PYTHON

- All class definitions start with the `class` keyword, which is followed by the name of the class and a colon (:).
 - By convention Python class names are written in CapitalizedWords notation, such as `Cinema`, `WineGlass`, `OfficeForIT`.
- The properties that all `Dog` objects must have are defined in a method called `__init__()`.
 - Every time a new `Dog` object is created, `__init__()` sets the initial **state** of the object by assigning the values of the object's properties. That is, `__init__()` initializes each new instance of the class.
 - You can give `__init__()` any number of parameters, but the first parameter will always be a variable called `self`.
 - When a new class instance is created, the instance is automatically passed to the `self` parameter in `__init__()` so that new attributes can be defined on the object.

```
class Dog:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

INSTANTIATE AN OBJECT IN PYTHON

- Creating a new object from a class is called instantiating an object.
- You can instantiate a new Dog object by typing the name of the class, followed by opening and closing parentheses () .

```
d1 = Dog("Rockie", 5)  
d2 = Dog("Coolman", 2)
```

- This creates two new Dog instances.
- When you instantiate a Dog object, Python creates a new instance and passes it to the first parameter of `__init__()`.
- This essentially removes the `self` parameter, so you only need to worry about the name and age parameters.

ACCESS INSTANCE ATTRIBUTES

- Instance attributes are accessed using
dot notation:

d1.name Rockie

d1.age 5

d2.name Coolman

d2.age 2

INSTANCE METHODS

- Instance methods are functions that are *defined inside a class* and can only be called from an instance of that class.
- This Dog class has two instance methods:
 - `.description()`

returns a string displaying the name and age of the dog.

- `.eat()`

has one parameter called food and returns a string containing the dog's name and the food the dog eats.

```
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def eat(self, food):
        return f"{self.name} eats {food}"
```

```
d1 = Dog("Rockie", 5)
```

```
d1.description()
```

```
'Rockie is 5 years old'
```

```
d1.eat("pork")
```

```
'Rockie eats pork'
```

```
d1.eat("snacks")
```

```
'Rockie eats snacks'
```

LECTURE 3

PERFORMANCE ANALYSIS

SEHH2239 Data Structures

LEARNING OBJECTIVES:

- To understand the use of algorithm analysis
- To assess the efficiency of a given algorithm
- To compare the expected execution time of different algorithms

ALGORITHMS AND PSEUDO CODES

WHAT IS AN ALGORITHM?

- An algorithm is a step by step method of solving a problem.
 - E.g. Find the path from home to the campus. Cook a streamed fish, etc.
- It is commonly used for data processing, calculation and other related computer and mathematical operations.*



*Quote from: <https://www.techopedia.com/definition/3739/algorithm>

PSEUDOCODE

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

ALGORITHMS AND PSEUDOCODES

- Pseudocode is a *step-by-step written outline* of your code that you can gradually transcribe into the programming language.
- *Describing how an algorithm should work.*
- Pseudocode can illustrate where a particular construct, mechanism, or technique could or must appear in a program.

PSEUDOCODE AND ACTUAL CODE

Pseudocode

```
If age is 65 or above  
    Group is senior  
Else if age is 18 or above  
    Group is adult  
Else  
    Group is children  
Display Group
```

In JAVA

```
if (age >= 65)  
    group = "senior";  
  
else if (age >= 18)  
    group = "adult";  
else  
    group = "children";  
System.out.println(group);
```

Actual Code

In PYTHON

```
if age >= 65:  
    group = "senior"  
else:  
    if age >= 18:  
        group = "adult"  
    else:  
        group = "children"  
print(group)
```

PSEUDOCODE DETAILS

- Control flow
 - **if ... then ... [else ...]**
 - **while ... do ...**
 - **repeat ... until ...**
 - **for ... do ...**
 - Indentation replaces braces
- Method declaration

Algorithm *method* (*arg* [, *arg*...])

Input ...

Output ...
- Method call

var.method (*arg* [, *arg*...])
- Return value

return *expression*
- Expressions
 - ←Assignment
(like = in Python)
 - = Equality testing
(like == in Python)

*n²*Superscripts and other mathematical formatting allowed

EFFICIENCY OF ALGORITHMS

ALGORITHM EFFICIENCY

Problem:

$$\text{Total} = 1+2+3\dots+N.$$

where N is any +ve integer

Find Total.

Algorithm B

```
total = 0  
for i : 1 to N  
    for m : 1 to i  
        total = total + 1
```

Algorithm A

```
total = 0  
for i : 1 to N  
    total = total + i
```

Algorithm C

```
total = N * ( N + 1 ) / 2
```

- Which one runs fastest? Which slowest?

MEASURING AN ALGORITHM EFFICIENCY

- How to measure efficiency to compare different algorithms to solve a problem?
- The process of measuring the *complexity* of algorithms is called ***analysis of algorithms***.

Complexity

- Time Complexity – The time it takes to execute
- Space Complexity – The memory it needs to execute
- Each of them can be analyzed separately.
 - Focus on the time complexity of algorithms.
 - As more important, and memory size grows exponentially.
 - Inverse relation between time and space required.

PROBLEM SIZE

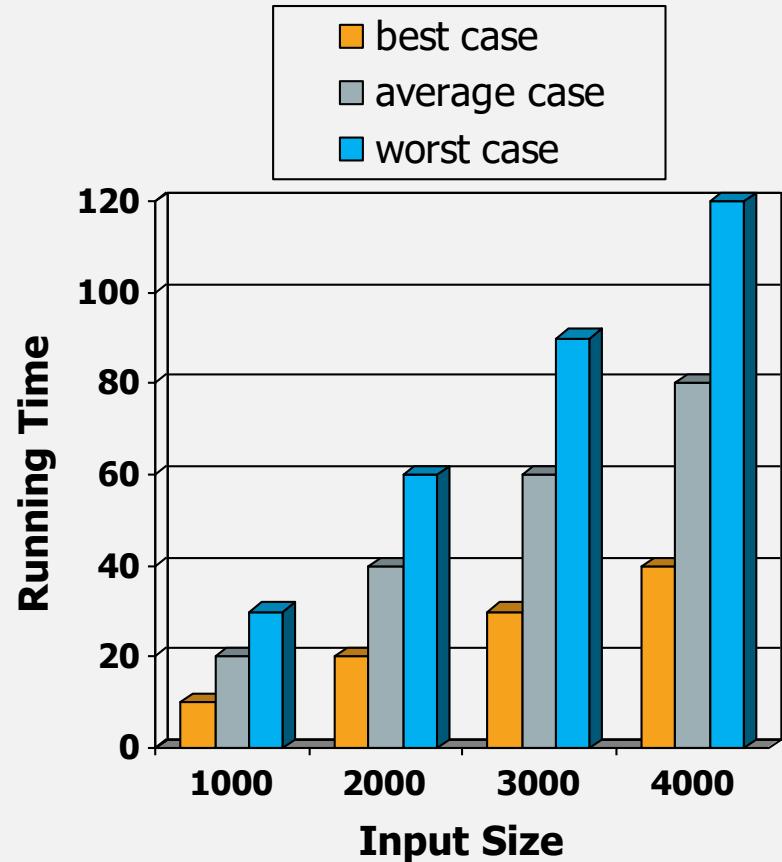
- Problem size is the *number of items* that an algorithm processes.
 - E.g. Number of elements in a list.
- The *running time* of an algorithm typically grows with the input size.
 - E.g. Time needed for removing a elements grows with the number of elements in a list.

BEST, WORST AND AVERAGE CASES OF RUNNING TIME

- Best-case
 - The algorithm takes the least time and it can do no better than that.
- Worst-case
 - The algorithm takes the most time and it can do no worse than that.
- Average-case
 - The average case on take typical data.

RUNNING TIME

- Average case time is often difficult to determine.
- We focus on the **worst case** running time.
 - Easier to analyze
 - Crucial to applications such as games, finance and robotics
 - Worst-case count = maximum count



BIG-OH NOTATION

BIG-OH NOTATION

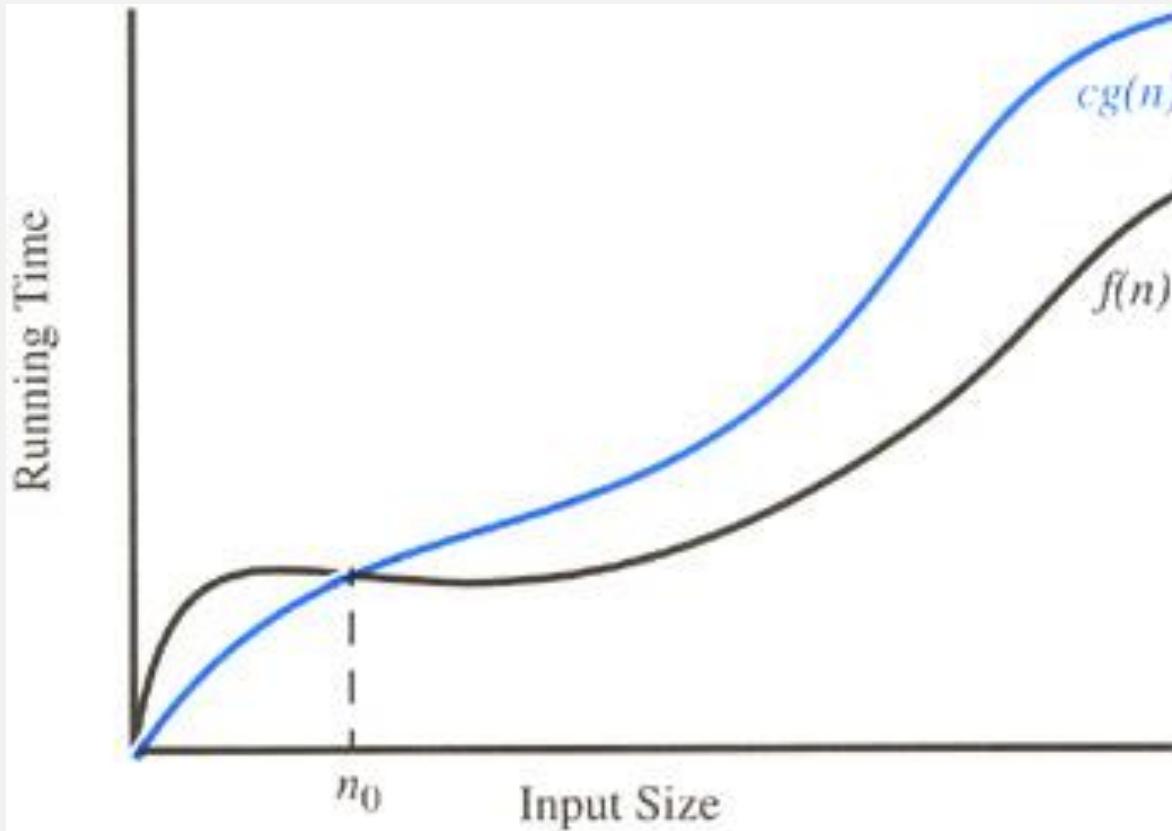
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

$c \times g(n)$ gives the **upper-bound** on $f(n)$.

$$f(n) \leq O(g(n))$$

BIG-OH NOTATION



Illustrating the "big-Oh" notation. The function $f(n)$ is $O(g(n))$,
for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$

BIG-OH NOTATION

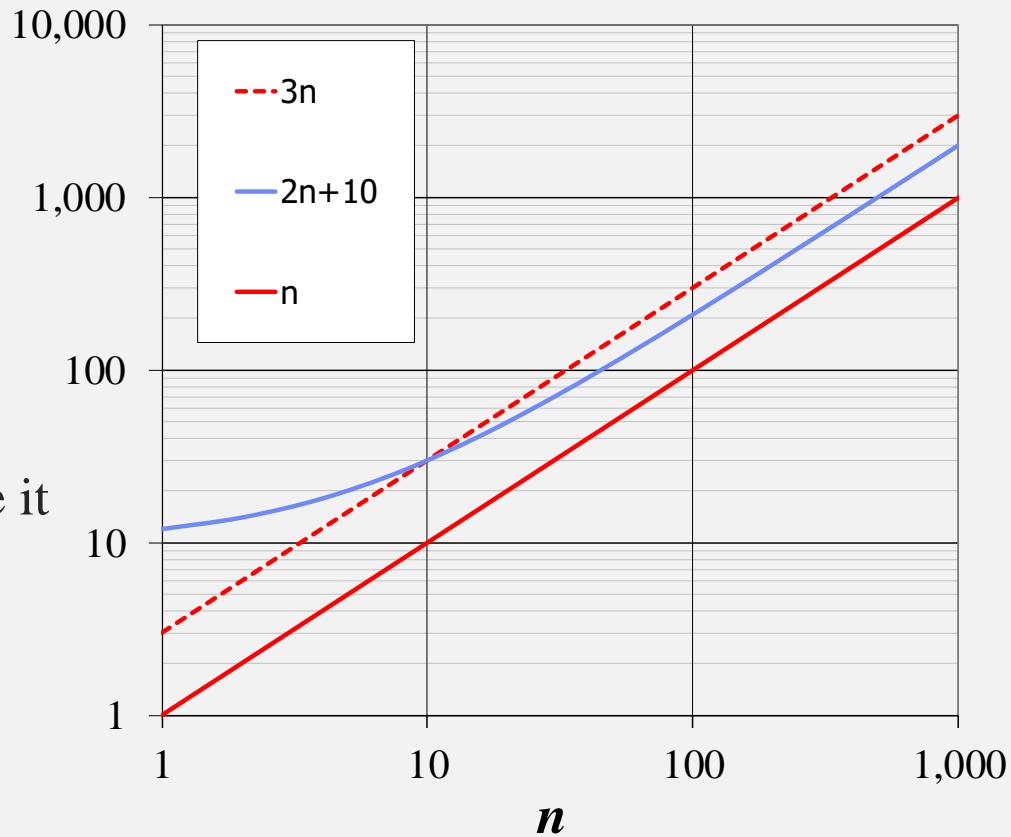
Example:

- $2n + 10$ is $O(n)$

Because $2n + 10 \leq cn$

for $c = 3$ and $n_0 = 10$

(That is, there are positive constants c and n_0 to make it true.



BIG-OH NOTATION

O(n)

- We read $O(n)$ as either “Big Oh of n ” or “order of at most n ”.

Example

$$\begin{cases} f(n) = 6n + 3 \\ g(n) = n \end{cases}$$

$$f(n) \leq 7n \text{ for } n \geq n_0 \geq 2$$

- If an algorithm uses $6n+3$ operations, it requires time proportional to n . We say it is $O(n)$.
- If an algorithm has a time requirement of proportional to n^2 , we say that it is $O(n^2)$.

MORE BIG-OH EXAMPLES

□ $[7n - 2 \leq 7n]_{n \geq n_0 = 1}$
7n-2 is O(n) $c=7$ O(n)

need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq c n$ for $n \geq n_0$

this is true for $c = 7$ and $n_0 = 1$

□ $[3n^3 + 20n^2 + 5n^0 \leq 3n^3 + 20n^3 + 5n^3]$
3 $n^3 + 20 n^2 + 5$ is O(n^3) $= 28n^3$ for $n \geq 1$

need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c n^3$ for $n \geq n_0$

this is true for $c = 4$ and $n_0 = 21$

□ $[3 \log n + 5 \leq 3 \log n + 5 \lg n]_{n \geq n_0 = 2}$
3 $\log n + 5$ is O($\log n$) $c = 8$ O($\lg n$)

need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

this is true for $c = 8$ and $n_0 = 2$

BIG-OH RULES

- If $f(n)$ is a polynomial of degree d , then $f(n)$ is $O(n^d)$, i.e.,

1. Drop lower-order terms

2. Drop constant factors

$$ax^3 + bx^2 + cx + d \quad \cancel{x^3} \quad \cancel{x^2} \quad \cancel{x} \quad \cancel{d}$$

- Use the **smallest possible** class of functions

• ✓ “ $2n$ is $O(n)$ ” ✗ $O(n^2)$

$O(n)$

- Use the **simplest expression** of the class

• ✓ “ $3n + 5$ is $O(n)$ ” ✗ $O(3n)$

BIG-OH EXAMPLES

$20n^3 + 10n\log n + 5$ is $O(n^3)$.

Justification: $20n^3 + 10n\log n + 5 \leq 35n^3$, for $n \geq 1$.

In fact, any polynomial $a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$ will always be $O(n^k)$.

$3 \log n + \log \log n$ is $O(\log n)$.

Justification: $3 \log n + \log \log n \leq 4 \log n$, for $n \geq 2$. Note that $\log \log n$ is not even defined for $n = 1$. That is why we use $n \geq 2$.

2^{100} is $O(1)$.

Justification: $2^{100} \leq 2^{100} \cdot 1$, for $n \geq 1$. Note that variable n does not appear in the inequality, since we are dealing with constant-valued functions.

$5/n$ is $O(1/n)$.

Justification: $5/n \leq 5(1/n)$, for $n \geq 1$ (even though this is actually a decreasing function).

FIND THE BIG-OHS

$$10n + 7$$

$$100n^3 - 3$$

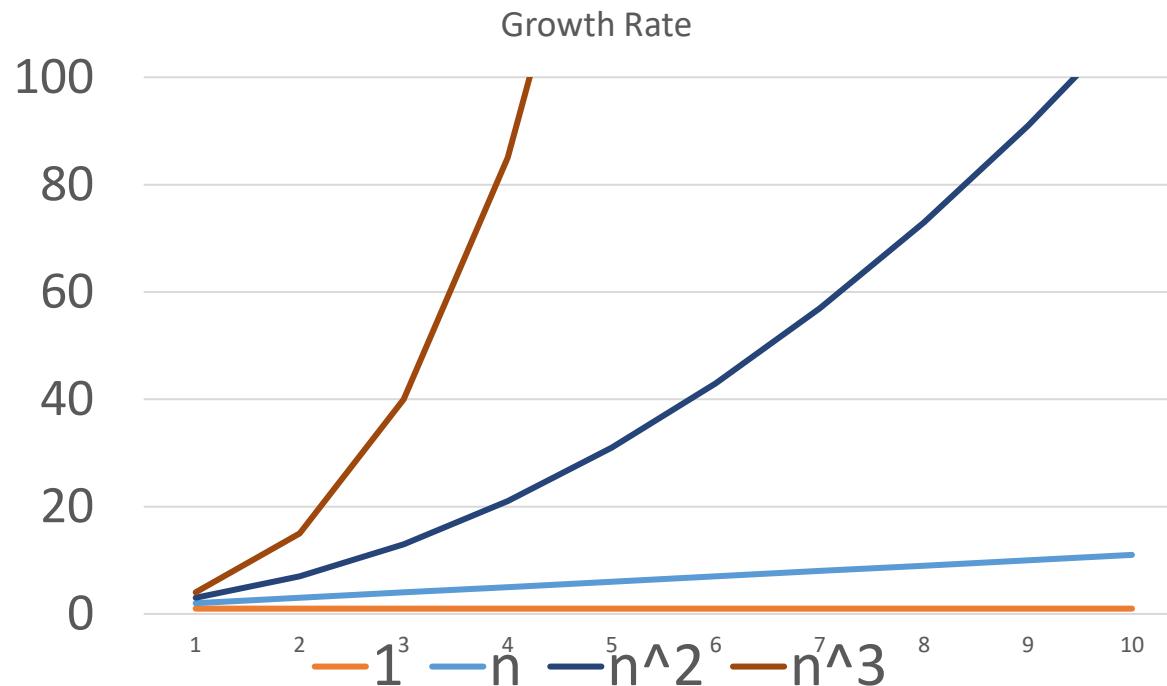
$$3n^2 + 2n + 6$$

$$8n^4 + 9n^2$$

GROWTH RATE FUNCTIONS

GROWTH RATE

- Relation to the problem size, n
- We care about n to be very large.
- Growth rate :



GROWTH RATE

- Functions

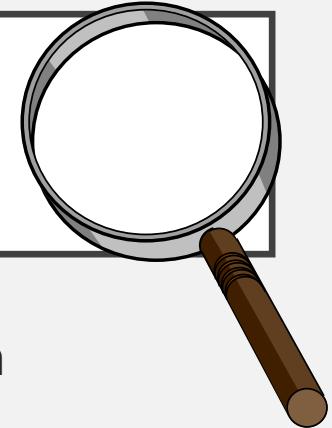
$$1 < \log(\log n) < \log(n) < n$$

$$n < n \log n < n^2 < n^3 < 2^n < n!$$

- Note that log here are base 2

ANALYSIS OF ALGORITHM EFFICIENCY IN BIG-OH NOTATION

THEORETICAL ANALYSIS



- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

ESTIMATING RUNNING TIME

- The meaning of Big-Oh functions in running time.
 - $O(1)$ is constant time.
 - $O(n)$ is linear time.
 - $O(n^2)$ is quadratic time.
 - $O(\log n)$ is logarithmic time.
 -
- Useful tricks:
 - Investigate nesting of loops due to inputs.
 - Look into the steps in a loop due to inputs.
- Focus in the worst case.

ALGORITHM EFFICIENCY

Problem:

$$\text{Total} = 1 + 2 + 3 \dots + N.$$

Find total.

Algorithm B

```
total = 0
for i : 1 to N
    for m : 1 to i
        total = total + 1
```

O(n^2)

Algorithm A

```
total = 0
for i : 1 to N
    total = total + i
```

O(n)

Algorithm C

```
total = N * ( N + 1 ) / 2
```

O(1)

EXAMPLE

```
for i in range(0, N):
    for j in range(N, i, -1):
        a = a + i + j
```

<https://realpython.com/lessons/time-complexity-overview/>

https://www.youtube.com/watch?v=5yJ_QLec0Lc

COMPLEXITY OF PYTHON LIST

- The complexity of Python List operations are listed in Table below.

Operation	Example	Complexity Class	Notes
Index	<code>l[i]</code>	$O(1)$	
Store	<code>l[i] = 0</code>	$O(1)$	
Length	<code>len(l)</code>	$O(1)$	
Append	<code>l.append(5)</code>	$O(1)$	
Pop	<code>l.pop()</code>	$O(1)$	same as <code>l.pop(-1)</code> , popping at end
Clear	<code>l.clear()</code>	$O(1)$	similar to <code>l = []</code>
Slice	<code>l[a:b]</code>	$O(b-a)$	$ [1:5]:O(1) [:]:O(\text{len}(l)-0)=O(N)$
Extend	<code>l.extend(...)</code>	$O(\text{len}(...))$	depends only on len of extension
Construction	<code>list(...)</code>	$O(\text{len}(...))$	depends on length of ... iterable
check ==, !=	<code>l1 == l2</code>	$O(N)$	
Insert	<code>l[a:b] = ...</code>	$O(N)$	
Delete	<code>del l[i]</code>	$O(N)$	
Containment	<code>x in/not in l</code>	$O(N)$	searches list
Copy	<code>l.copy()</code>	$O(N)$	Same as <code>l[:]</code> which is $O(N)$
Remove	<code>l.remove(...)</code>	$O(N)$	
Pop	<code>l.pop(i)</code>	$O(N)$	$O(N-i): l.pop(0):O(N)$ (see above)
Extreme value	<code>min(l)/max(l)</code>	$O(N)$	searches list
Reverse	<code>l.reverse()</code>	$O(N)$	
Iteration	<code>for v in l:</code>	$O(N)$	

SUMMARY

- An algorithm's complexity is described in terms of the time and space required to execute it.
- Compare efficiency of algorithms
- Big-Oh Notation
- Growth-rate function



LECTURE 4: RECURSION AND SORTING I

SEHH2239 Data Structures

Learning Objectives:

- To read, write and apply ***recursion methods***
- To implement sorting methods of ***insertion sorts, bubble sorts*** and ***selection sorts***
- To estimate the ***time efficiency*** of the above sorting methods

What is Recursion?

- Recursion is a problem-solving process that breaks a problem into *identical* but *smaller* problems.
- A method that calls itself is a ***recursive method***. The invocation is a ***recursive call***.
- ***Recursion vs Iteration***
 - A recursion method calls itself.
 - An iteration method contains a loop.
 - E.g. For loop, While loop.

A Simple Example of Recursion

- Classic example--the factorial function:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n - 1) \cdot n$$

- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot f(n-1) & \text{else} \end{cases}$$

As a Python method:

```
def factorial(x):  
    if x == 1:  
        return 1  
    else:  
        return (x * factorial(x-1))
```

Designing Linear Recursion

1. **Base case:** we must always have some base cases, which can be solved with recursion
 - e.g. $n == 0$
2. **Making progress:** for the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case
 - e.g. $n - 1$

□ Test for base cases

- Begin by testing for a set of base cases (there should be *at least one*).
- Every possible chain of recursive calls **must** eventually reach a base case, and the handling of each base case should not use recursion.

Tracing a Recursive Method

```
def calcFactorial(n):  
    if n == 1:  
        return 1  
    else:  
        return n * calcFactorial(n - 1)
```

```
def calcFactorial(n):  
    if n == 1:  
        return 1  
    else:  
        return n * calcFactorial(n - 1)
```

```
def calcFactorial(n):
```

```
    if n == 1:
```

```
        return 1
```

```
    else:
```

```
        return n * calcFactorial(n - 1)
```

3 * 2 * 1

3

3 * (2 * 1)

```
def calcFactorial(n):
```

```
    if n == 1:
```

```
        return 1
```

```
    else:
```

```
        return n * calcFactorial(n - 1)
```

2

2 * (1)

```
def calcFactorial(n):
```

```
    if n == 1:
```

```
        return 1
```

```
    else:
```

```
        return n * calcFactorial(n - 1)
```

1

Example of Linear Recursion (for further reading)

Algorithm `linearSum(A, n)`:

Input:

Array, A, of integers

Integer n such that

$0 \leq n \leq |A|$

Output:

Sum of the first n integers in A

if $n = 0$ then

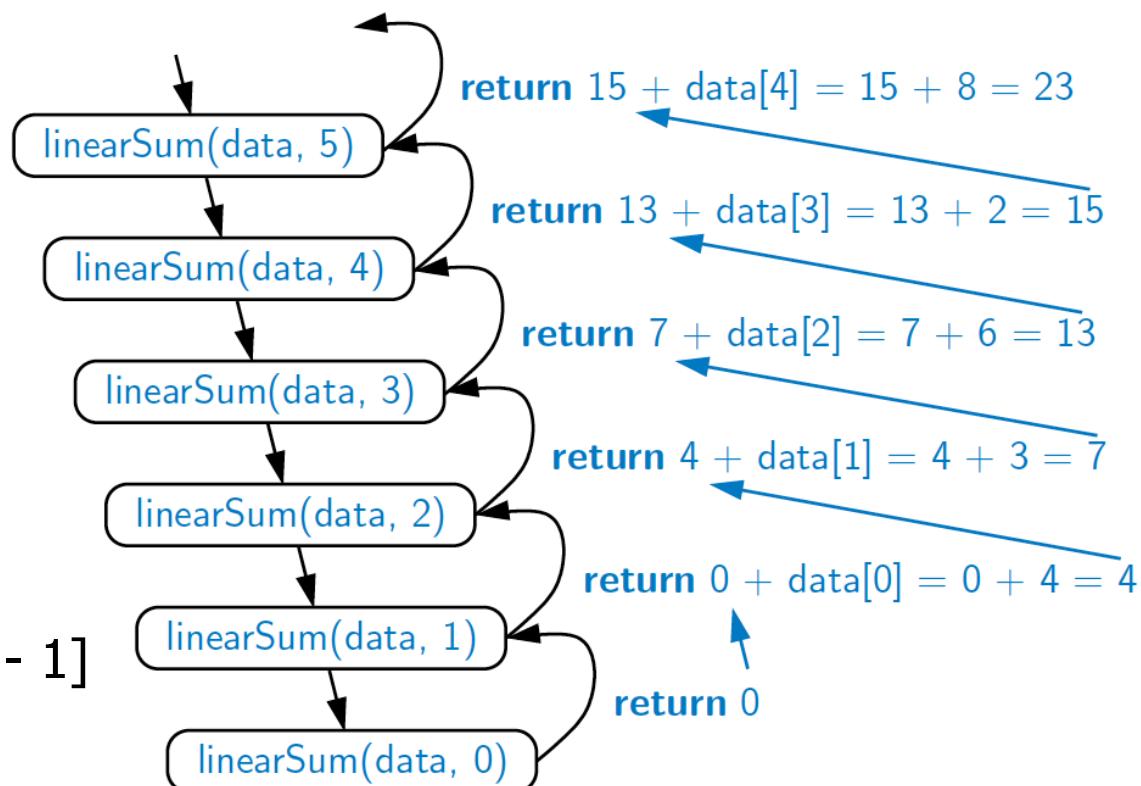
return 0

else

return

`linearSum(A, n - 1) + A[n - 1]`

Recursion trace of `linearSum(data, 5)`
called on array data = [4, 3, 6, 2, 8]



Reversing an Array (for further reading)

Algorithm **reverseArray**(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

if $i < j$ then

 Swap $A[i]$ and $A[j]$

reverseArray(A, $i + 1$, $j - 1$)

return



COMPARING DATA ITEMS

Equality: “is” and “==”

- Both “is” and “==” are used for object comparison in Python.
- “==” compares values of two objects,
- “is” checks if two objects are same
 - i.e. two references to same object.

```
a = [1, 2, 3]  
b = [1, 2, 3]
```

```
print(id(a))  
print(id(b))  
  
# "is" operator  
if a is b:  
    print("Same objects")  
else:  
    print("Different objects")
```

140031399513936
140031399518448

Different objects

“is” and “==”

- Further Examples

```
# "==" operator
if a == b:
    print("Same value")
else:
    print("Not the same value")
```

Same value

```
c = b
if c is b:
    print("Same objects")
else:
    print("Different objects")
```

Same objects

```
d = list(a)
print(d is a)
print(d == a)
```

False

True



COMPARING OBJECTS IN CLASS

Object Equal

- Python automatically calls the `__eq__` method of a class when you use the `==` operator to compare the instances of the class.

```
class Student:  
    def __init__(self, name, age, mark):  
        self.name = name  
        self.age = age  
        self.dsa_score = mark  
  
    def __eq__(self, other):  
        return self.dsa_score == other.dsa_score  
  
joe = Student("Joe Elipse", 20, 80)  
pally = Student("Pally Kueng", 30, 75)  
martin = Student("Martin Ip", 25, 75)  
  
print(joe == pally)          False  
print(pally == martin)      True
```

Overloaded behaviour of operators

- Python has magic methods to define overloaded behaviour of operators.
- The comparison operators (<, <=, >, >=, == and !=) can be overloaded by providing definition to `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__` and `__ne__` magic methods

```
class Student2:  
    def __init__(self, name, age, mark):  
        self.name = name  
        self.age = age      Click to add text  
        self.dsa_score = mark  
  
    def __eq__(self, other):  
        if isinstance(other, Student2):  
            return self.dsa_score == other.dsa_score  
  
    def __lt__(self, other):  
        if isinstance(other, Student2):  
            return self.dsa_score < other.dsa_score  
  
    def __le__(self, other):  
        if isinstance(other, Student2):  
            return self.dsa_score <= other.dsa_score
```

operator. `__lt__`(a, b)
operator. `__le__`(a, b)
operator. `__eq__`(a, b)
operator. `__ne__`(a, b)
operator. `__ge__`(a, b)
operator. `__gt__`(a, b)

Overloaded behaviour of operators

```
joey = Student2("Joey See", 20, 80)
cally = Student2("Cally Peng", 30, 75)
tom = Student2("Tom Miu", 25, 75)

print(joey == cally)           False
print(cally == tom)           True
print(joey < cally)           False
print(cally < tom)            False

print(joey <= cally)          False
print(cally <= tom)           True
```

SORTING I

What is Sorting?

- Sorting means to put data in order.
- **Ascending** from lowest to highest
- **Descending** from highest to lowest
- E.g. Rearrange $a[0], a[1], \dots, a[n-1]$ into ascending order. When done, $a[0] \leq a[1] \leq \dots \leq a[n-1]$
- $8, 6, 9, 4, 3 \Rightarrow 3, 4, 6, 8, 9$

Sorting Algorithms

- A sorting algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements.
- Simple Sorting Algorithms:
 - Insertion Sort
 - Selection Sort
 - Bubble Sort

INSERTION SORT

Algorithm of insertion sort

For each $a[i]$ in an array a of size n ,

- Compare $a[i]$ with all elements in the sorted sub-list
- Shift all the elements in the sorted sub-list that $> a[i]$
- Insert $a[i]$ to the place

Insert An Element

Insert

- Given a sorted list/sequence, insert a new element
- E.g. Given 3, 6, 9, 14 and Insert 5
 - Result 3, 5, 6, 9, 14

Steps:

Compare new element (5) and last one (14)

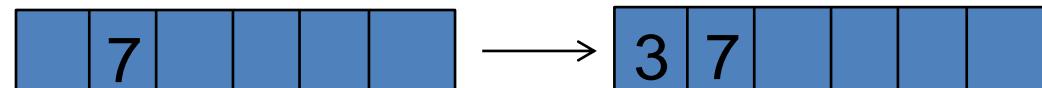
- Shift 14 right to get 3, 6, 9, , 14
- Shift 9 right to get 3, 6, , 9, 14
- Shift 6 right to get 3, , 6, 9, 14
- Insert 5 to get 3, 5, 6, 9, 14

Insertion Sort

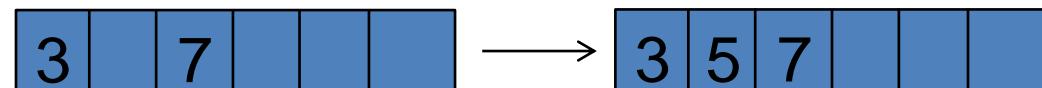
- Sort 7, 3, 5, 6, 1, 8



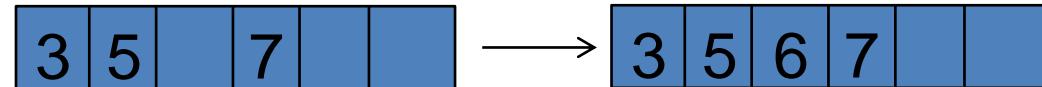
- Start with 7 and insert 3 => 3, 7



- Insert 5 => 3, 5, 7



- Insert 6 => 3, 5, 6, 7



- Insert 1 => 1, 3, 5, 6, 7



Insertion Sort in Python

```
def insertionSort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j] :
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
for i in range(len(arr)):

    print ("% d" % arr[i])
```

This code is contributed by Mohit Kumra

BUBBLE SORT

Algorithm of Bubble Sort

1. In an array a of size n , if $a[i-1] > a[i]$, swap $a[i-1]$ with $a[i]$ for $i=1, 2, 3, \dots n-1$. This is called a Bubbling Pass
2. After each bubbling pass, the largest element moves to the right-most position
3. Then, perform bubbling pass for the remaining $n-1$ elements, $a[0] \dots a[n-2]$, and obtain the largest at $a[n-2]$
4. And so on until performing bubbling pass for the remaining 2 elements and obtaining the largest at $a[1]$

Bubble Sort

First Bubbling Pass

- Sort 6, 5, 8, 4, 3, 1
- $6 > 5$, swap 6 and 5 => 5, 6, 8, 4, 3, 1
- $6 < 8$, no swap => 5, 6, 8, 4, 3, 1
- $8 > 4$, swap 8 and 4 => 5, 6, 4, 8, 3, 1
- $8 > 3$, swap 8 and 3 => 5, 6, 4, 3, 8, 1
- $8 > 1$, swap 8 and 1 => 5, 6, 4, 3, 1, 8
- After the **1st pass**, the largest element 8 moves to the right-most position

Bubble Sort

Second Bubbling Pass

- Sort the remaining 5, 6, 4, 3, 1
- $5 < 6$, no swap => 5, 6, 4, 3, 1
- $6 > 4$, swap 6 and 4 => 5, 4, 6, 3, 1
- $6 > 3$, swap 6 and 3 => 5, 4, 3, 6, 1
- $6 > 1$, swap 6 and 1 => 5, 4, 3, 1, 6
- After the **2nd pass**, the largest element 6 moves to the right-most position

Bubble Sort

Keep on Bubbling

- After the **3rd pass**, the largest element 5 moves to the right-most position, and the result is: 4, 3, 1, **5**
- After the **4th pass**, the result is: 3, 1, **4**
- After the **5th pass**, the result is: 1, **3**
- At the **6th pass**, one element remains, that is, **1**
- Combining $a[0]=1$, $a[1]=3$, ... $a[5]=8$ to get the sorted array **1, 3, 4, 5, 6, 8**.

Bubble Sort

```
def bubble_sort(our_list):
    # We go through the list as many times as there are elements
    for i in range(len(our_list)):
        # We want the last pair of adjacent elements to be (n-2, n-1)
        for j in range(len(our_list) - 1):
            if our_list[j] > our_list[j+1]:
                # Swap
                our_list[j], our_list[j+1] = our_list[j+1], our_list[j]

our_list = [19, 13, 6, 2, 18, 8]
bubble_sort(our_list)
print(our_list)
```

Early-Terminating Bubble Sort

Bubble Sort

- Sort 6, 1, 2, 5, 3, 4
- 1st pass => 1, 2, 5, 3, 4, 6
- 2nd pass => 1, 2, 3, 4, 5, 6
- 3rd pass, no swap => 1, 2, 3, 4, 5, 6
- 4th pass, no swap => 1, 2, 3, 4, 5, 6
- 5th pass, no swap => 1, 2, 3, 4, 5, 6 No need after this point
- 6th pass, no swap => 1, 2, 3, 4, 5, 6
- Terminate and get the sorted list

Early-Terminating Bubble Sort

- ▲ Sort 6, 1, 2, 5, 3, 4
- ▲ 1st pass => 1, 2, 5, 3, 4, 6
- ▲ 2nd pass => 1, 2, 3, 4, 5, 6
- ▲ 3rd pass, no swap => 1, 2, 3, 4, 5, 6
- ▲ Terminate and get sorted list



Early-Terminating Bubble Sort

- If a bubbling pass results in no swaps, then the array is in sorted order and no further bubbling passes are necessary.
- *Note: It needs a pass for the checking of order*

Early-Terminating Bubble Sort

```
def bubble_sort_earlyterm(our_list):
    has_swapped = True

    num_of_iterations = 0

    while(has_swapped):
        has_swapped = False
        for i in range(len(our_list) - num_of_iterations - 1):
            if our_list[i] > our_list[i+1]:
                # Swap
                our_list[i], our_list[i+1] = our_list[i+1], our_list[i]
                has_swapped = True
            num_of_iterations += 1

    our_list = [19, 13, 6, 2, 18, 8]
    bubble_sort_earlyterm(our_list)
    print(our_list)
```

#<https://stackabuse.com/bubble-sort-in-python/>

SELECTION SORT

Algorithm of Selection Sort

1. In an array a of size n , determine the largest element and swap the largest with the last element $a[n-1]$
2. Then, determine the largest of the remaining $n-1$ elements, $a[0] \dots a[n-2]$, and swap that largest with $a[n-2]$
3. And so on until determining the largest of the remaining 2 elements and swap the largest with $a[1]$

Selection Sort

- Sort 6, 5, 8, 4, 3, 1
- Largest is 8, swap with 1 => 6, 5, 1, 4, 3, 8
- Sort 6, 5, 1, 4, 3
- Largest is 6, swap with 3 => 3, 5, 1, 4, 6, 8
- Sort 3, 5, 1, 4
- Largest is 5, swap with 4 => 3, 4, 1, 5, 6, 8
- Sort 3, 4, 1
- Largest is 4, swap with 1 => 3, 1, 4, 5, 6, 8
- Sort 3, 1
- Largest is 3, swap with 1 => 1, 3, 4, 5, 6, 8

Selection Sort

```
def selection_sort(L):
    # i indicates how many items were sorted
    for i in range(len(L)-1):
        # To find the minimum value of the unsorted segment
        # We first assume that the first element is the lowest
        min_index = i
        # We then use j to loop through the remaining elements
        for j in range(i+1, len(L)):
            # Update the min_index if the element at j is lower than it
            if L[j] < L[min_index]:
                min_index = j
        # After finding the lowest item of the unsorted regions, swap with the first unsorted item
        L[i], L[min_index] = L[min_index], L[i]
```

```
L = [3, 1, 41, 59, 26, 53, 59]
print(L)
selection_sort(L) # Let's see the list after we run
the Selection Sort
print(L)
```

Early-Terminating Selection Sort

- **Shortcoming:** iteration keeps on even if the elements have been sorted
- Use Early-Terminating Selection Sort to eliminate unnecessary iterations
- In Early-Terminating Selection Sort, during the scan for the largest element, it checks to see whether the array is already sorted or not

Summary on Key terms

- **Recursion**
- **Insertion sort**
- **Bubble sort**
 - Early-Terminating
- **Selection sort**

LECTURE 5 : Sorting II

SEHH2239 Data Structures

Learning Objectives:

- To able to use the divide and conquer
- To describe and implement Merge sort and Quick sort



DIVIDE AND CONQUER

Divide and Conquer

Divide-and-conquer algorithms generally have best complexity when a large instance is divided into smaller instances of approximately the same size.

1. **Base Case**, solve the problem **directly** if it is small enough
2. **Divide** the problem into two or more **similar and smaller** subproblems
3. **Recursively solve** the subproblems
4. **Combine** solutions to the subproblems

Divide and Conquer - Sort

Problem:

- Input: $A[\text{left}..\text{right}]$ – **unsorted** array of integers
- Output: $A[\text{left}..\text{right}]$ – **sorted** in non-decreasing order

Examples are **Merge Sort** and **Quick Sort**

Divide and Conquer - Sort

1. Base case
at most one element ($\text{left} \geq \text{right}$), return
2. Divide A into two subarrays: FirstPart, SecondPart
Two Subproblems:
sort the FirstPart
sort the SecondPart
3. Recursively
sort FirstPart
sort SecondPart
4. Combine sorted FirstPart and sorted SecondPart

MERGE SORT

Merge sort

- Merge sort keeps on dividing the list into equal halves until it can no more be divided.
- By definition, if it is only one element in the list, it is sorted.
- Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Merge Sort

- First $\text{ceil}(n/2)$ elements define one of the smaller instances; remaining elements define the second smaller instance.
- Each of the **two smaller instances is sorted recursively.**
- The sorted smaller instances are **combined** using a process called **merge**.

$$\text{ceil}(n/2)$$

$$\left\lceil \frac{25}{2} \right\rceil = 13$$

$$\text{floor}(n/2)$$

$$\left\lfloor \frac{25}{2} \right\rfloor = 12$$

Merge Sort: Idea

Divide into
two halves

A

FirstPart

SecondPart



Recursively
sort

Merge



A is sorted!

Merge Sort: Algorithm

Merge-Sort (A, left, right)

if $\text{left} \geq \text{right}$ return

else

 middle $\leftarrow \lfloor (\text{left}+\text{right})/2 \rfloor$

Merge-Sort(A, left, middle)

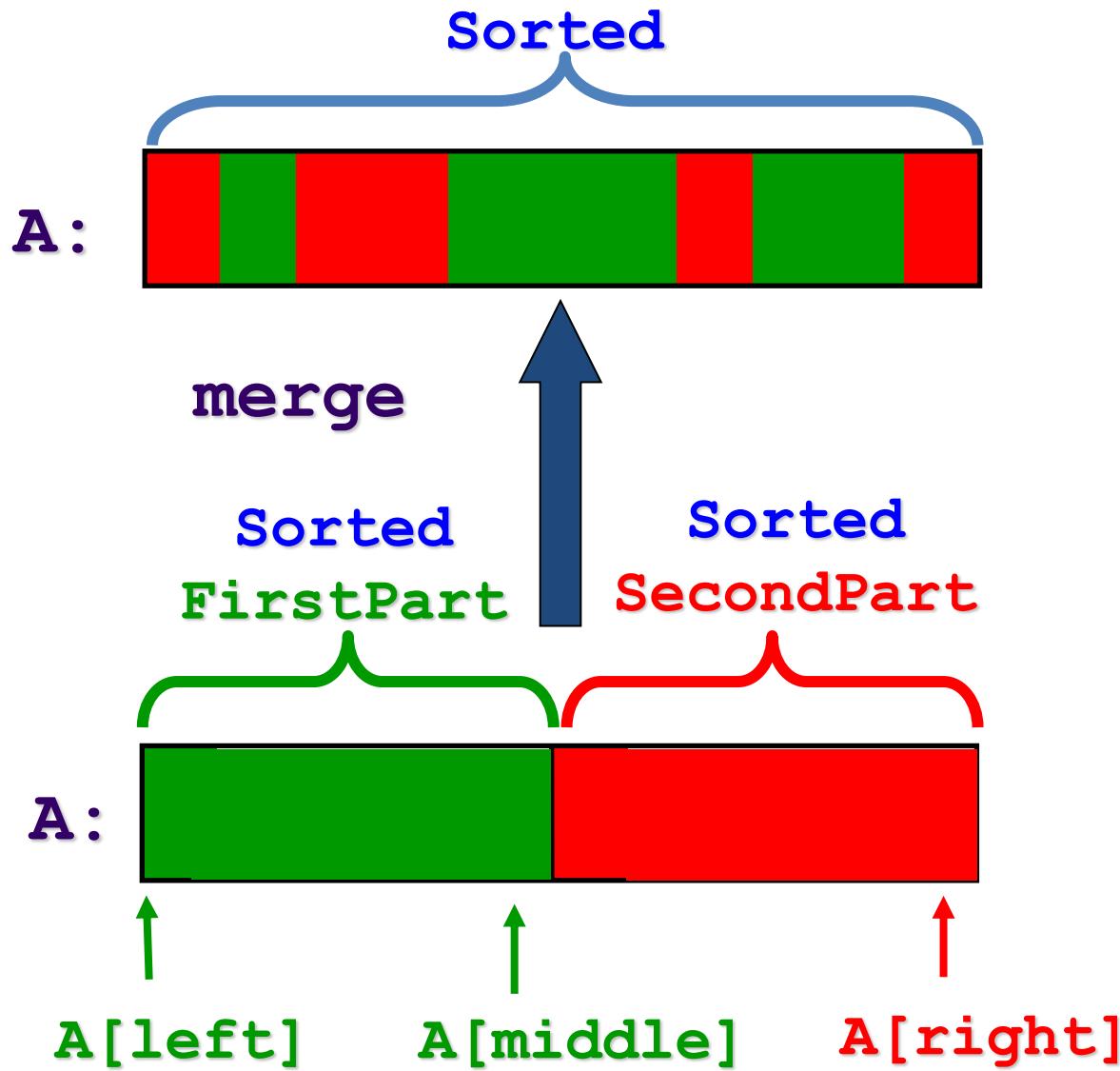
Recursive Call

Merge-Sort(A, middle+1, right)

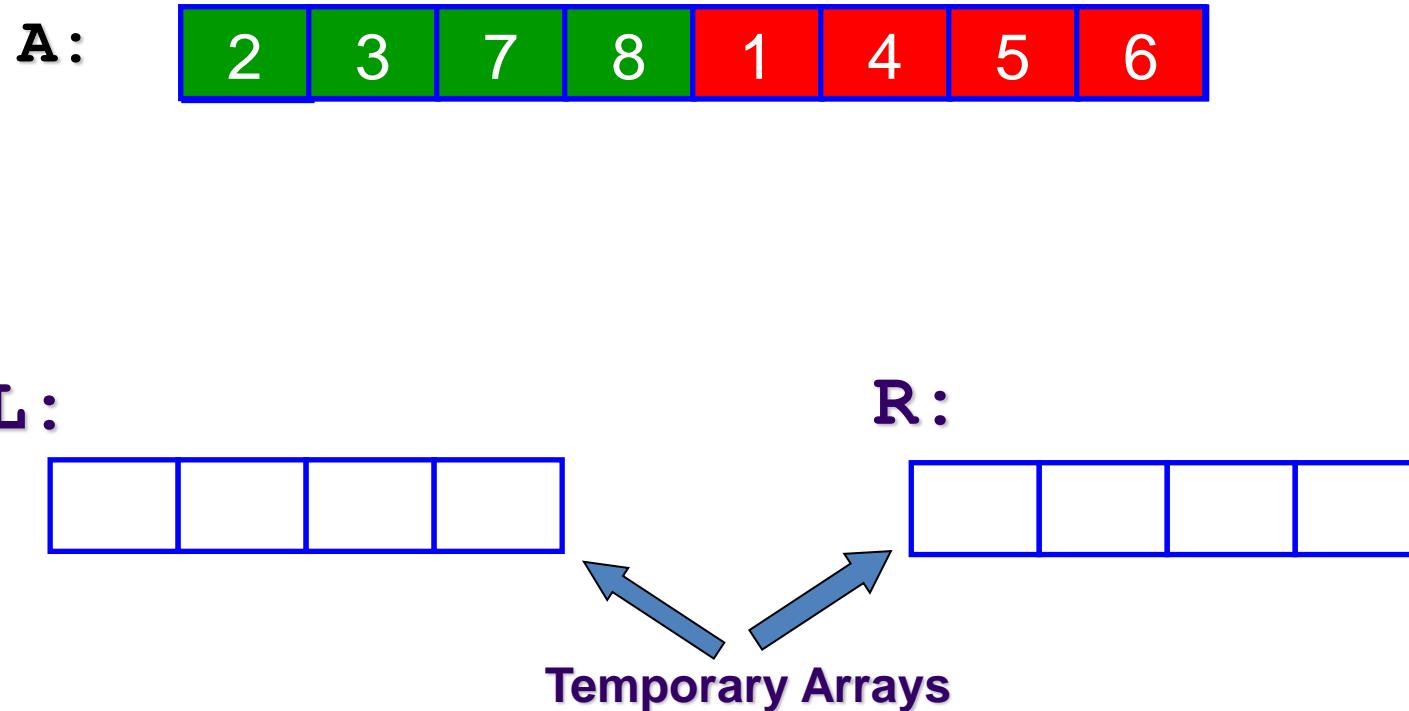
Merge(A, left, middle, right)

MERGING WITH ARRAY

Merge-Sort: Merge

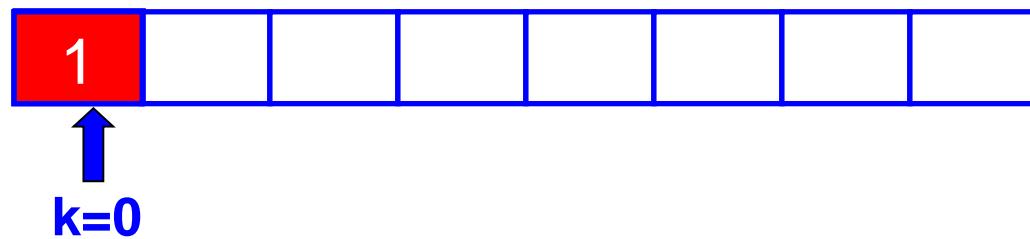


Merge-Sort: Merge Example

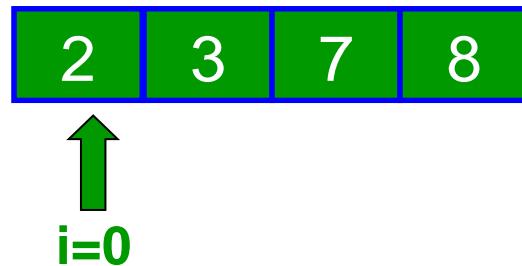


Merge-Sort: Merge Example

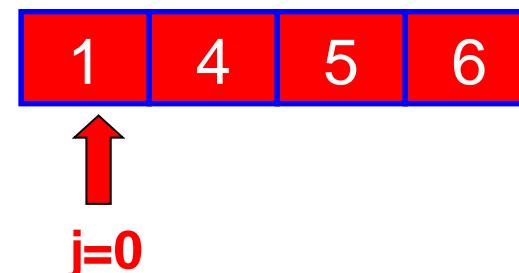
A:



L:

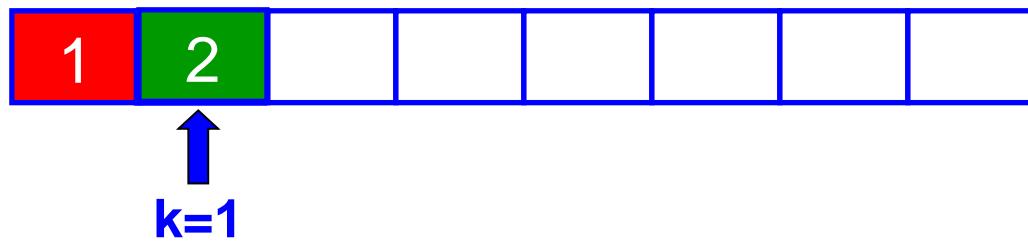


R:

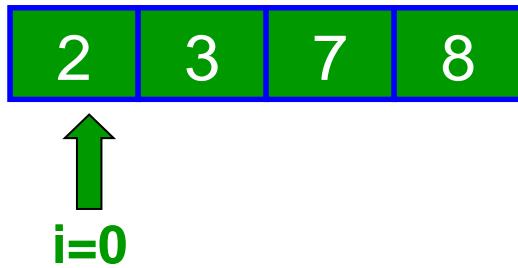


Merge-Sort: Merge Example

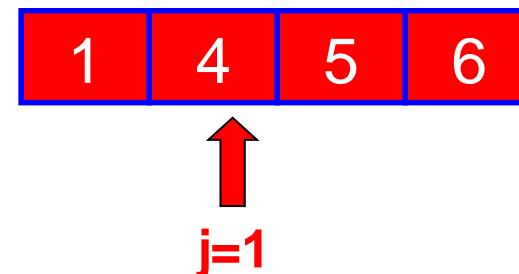
A:



L:

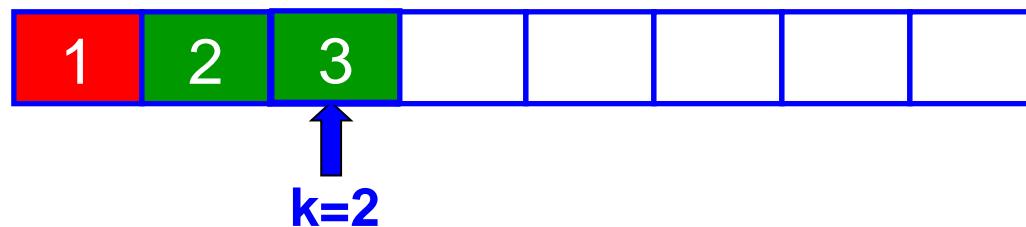


R:

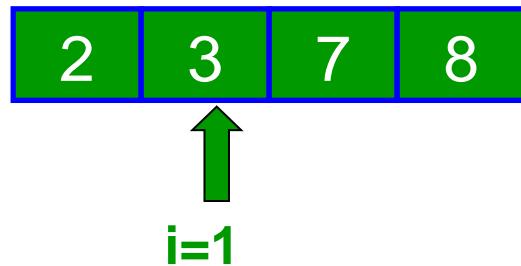


Merge-Sort: Merge Example

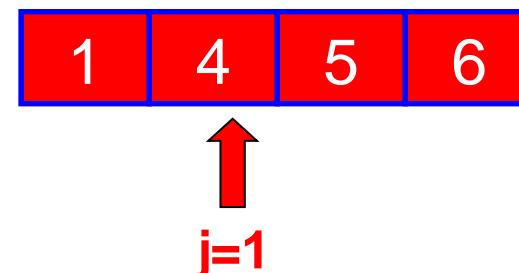
A:



L:

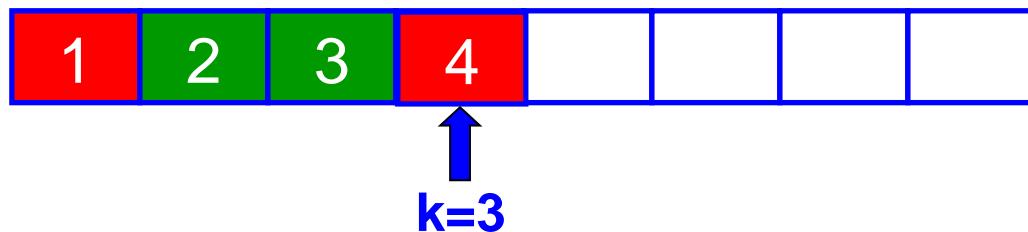


R:

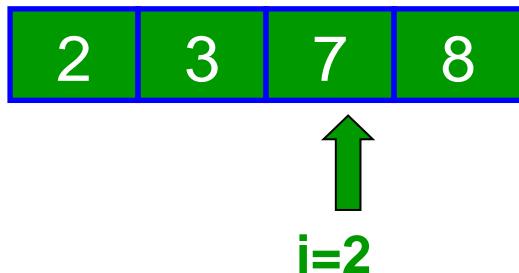


Merge-Sort: Merge Example

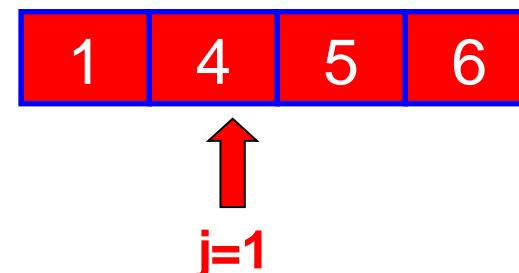
A:



L:

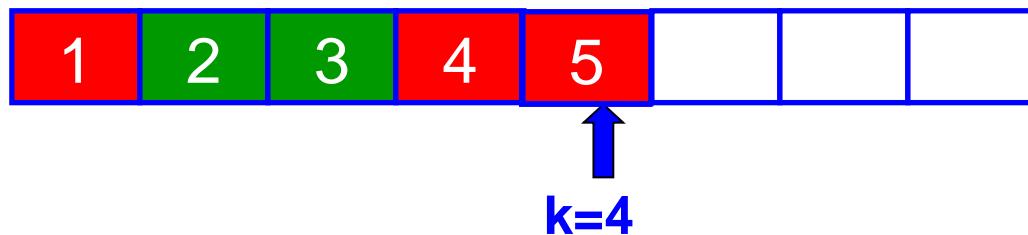


R:

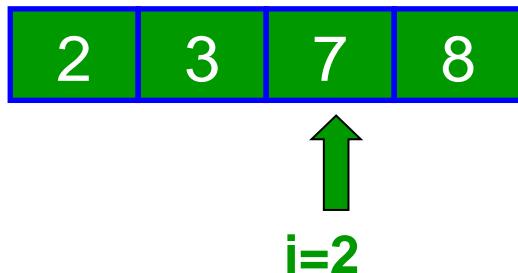


Merge-Sort: Merge Example

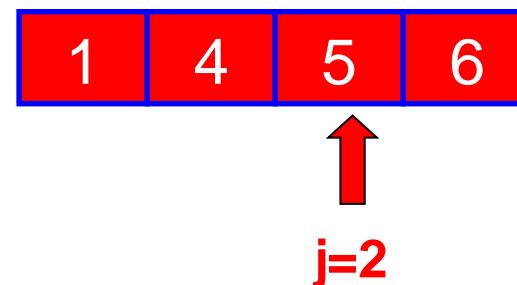
A:



L:

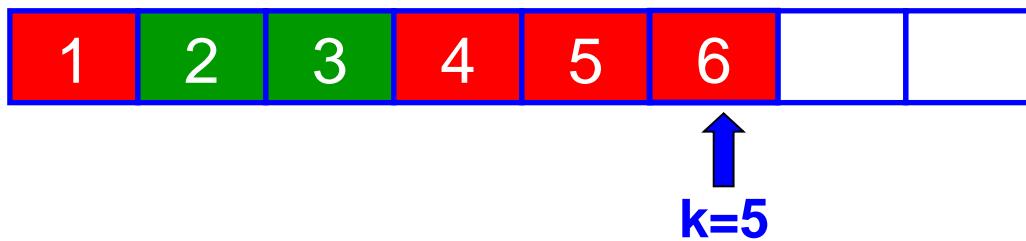


R:

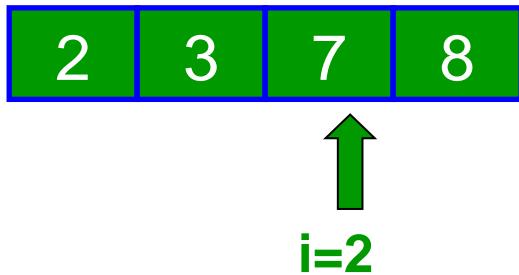


Merge-Sort: Merge Example

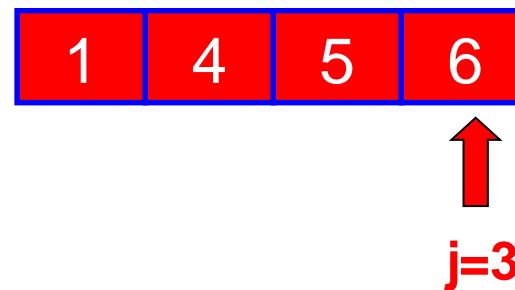
A:



L:

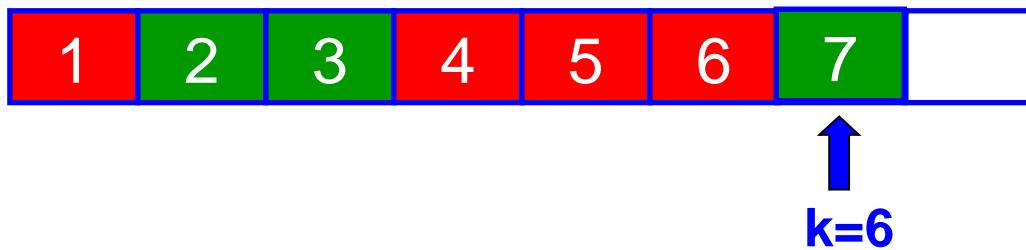


R:

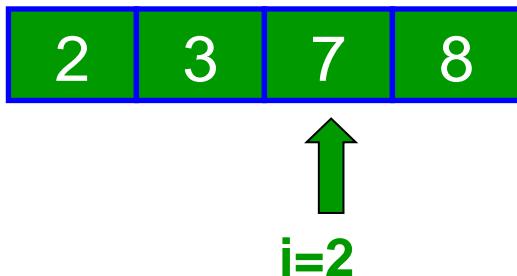


Merge-Sort: Merge Example

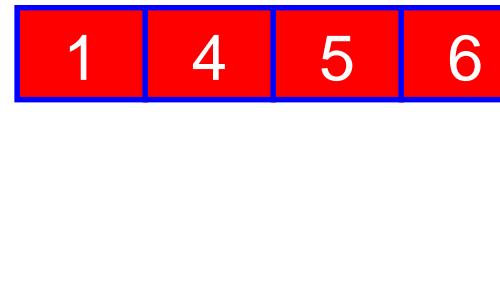
A:



L:

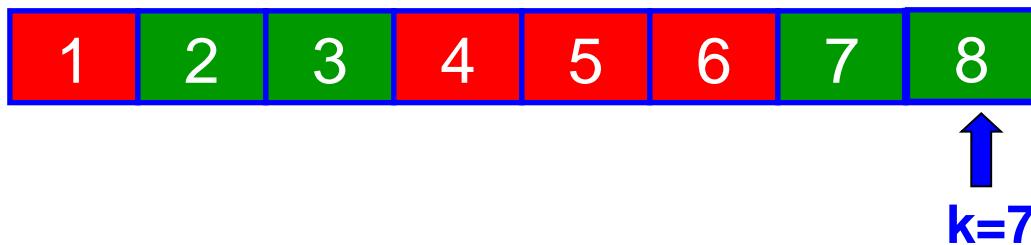


R:

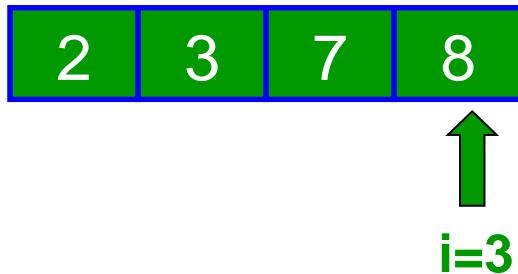


Merge-Sort: Merge Example

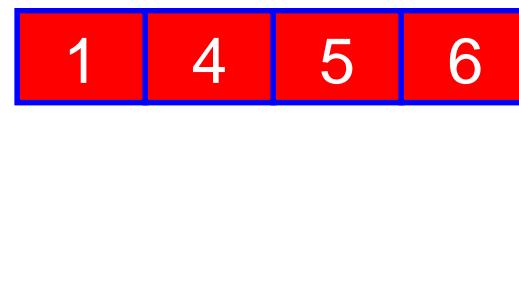
A:



L:



R:



Merge-Sort: Merge Example

A:



↑
k=8

L:



↑
i=4

R:



↑
j=4



MERGE SORT ILLUSTRATION

Merge-Sort(A, 0, 7)

Divide

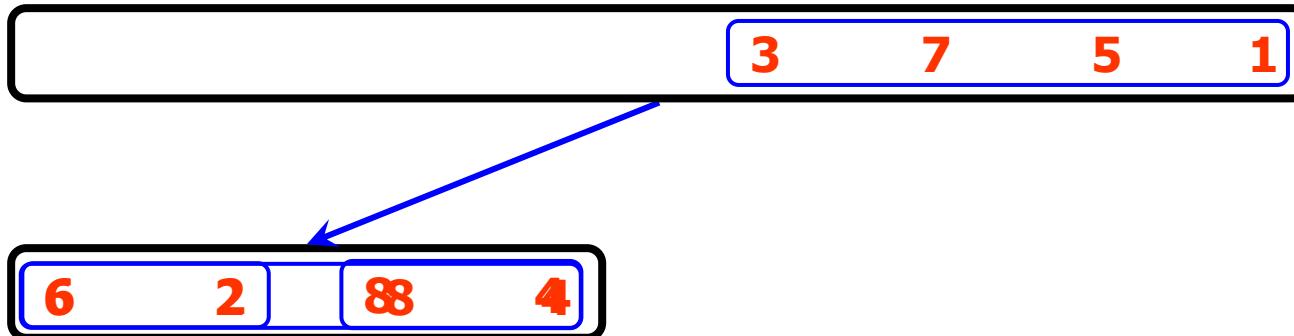
A:



Merge-Sort(A, 0, 7)

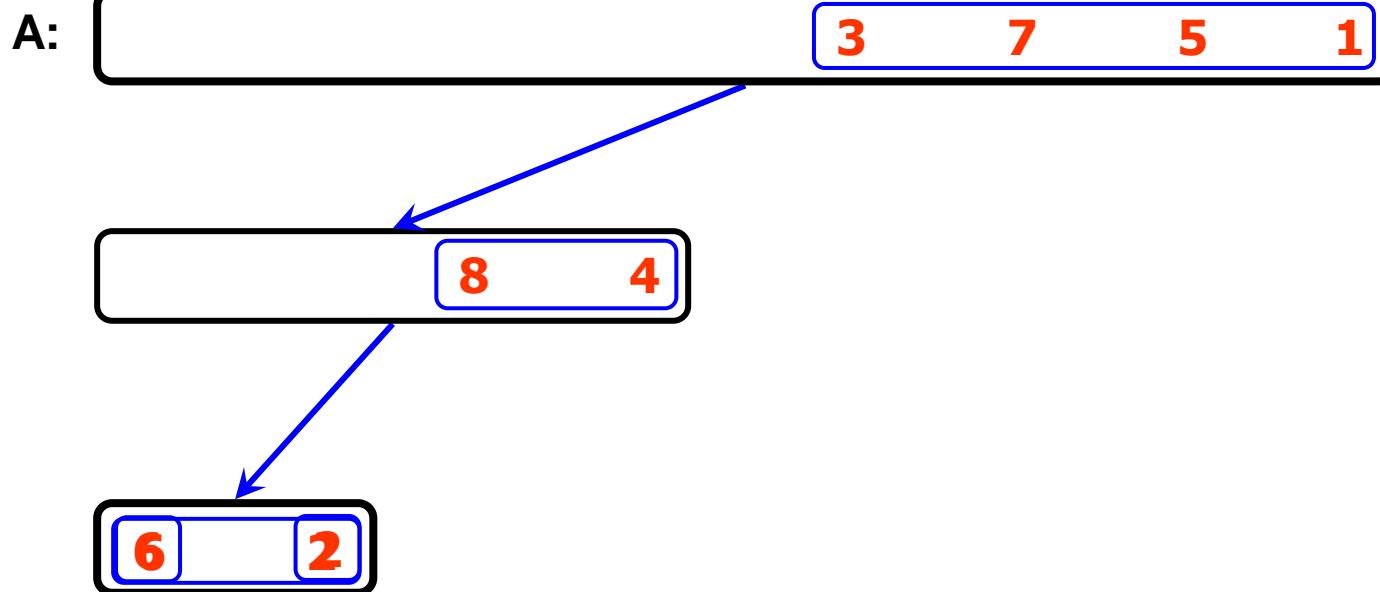
Merge-Sort(A, 0, 3), divide

A:



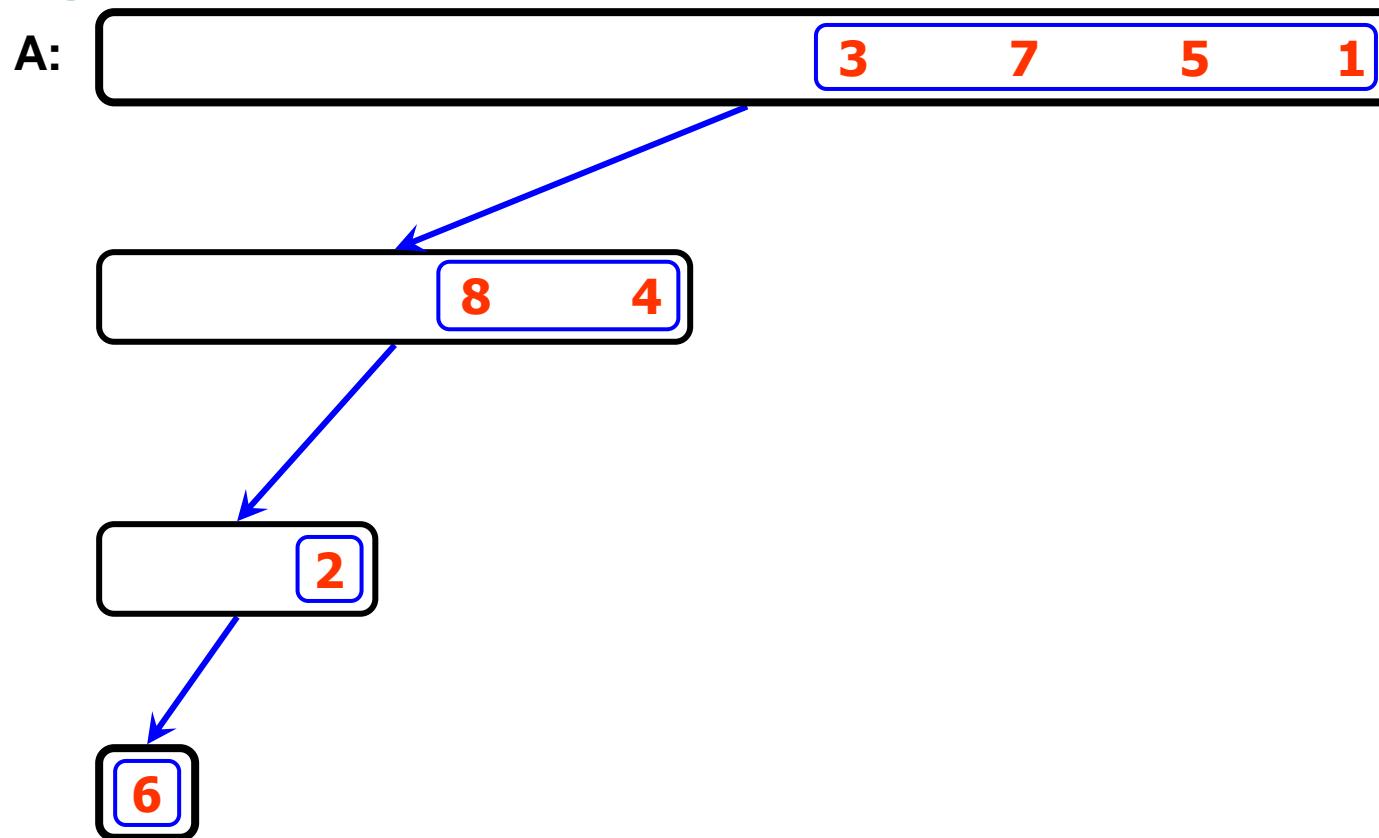
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 1), divide



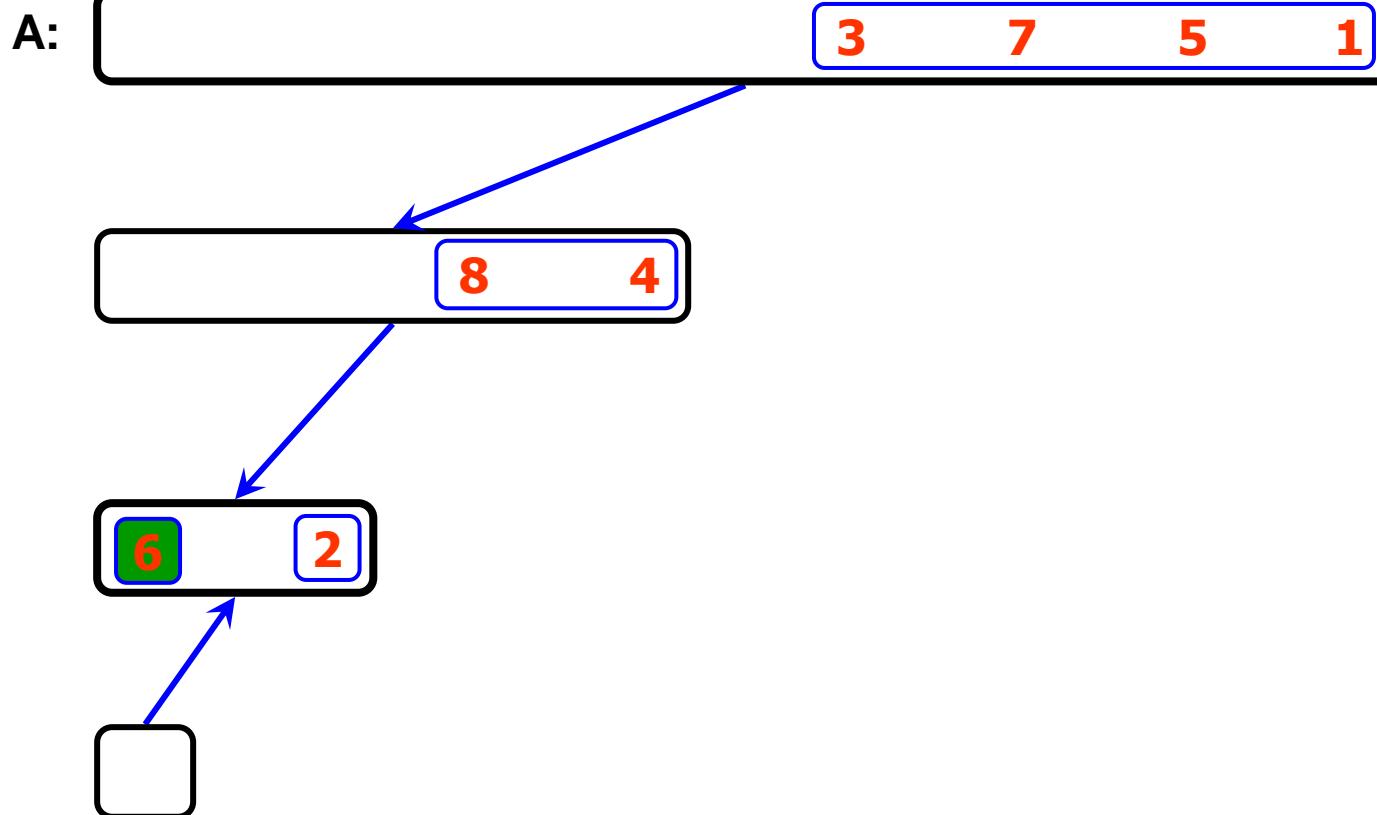
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 0) , base case



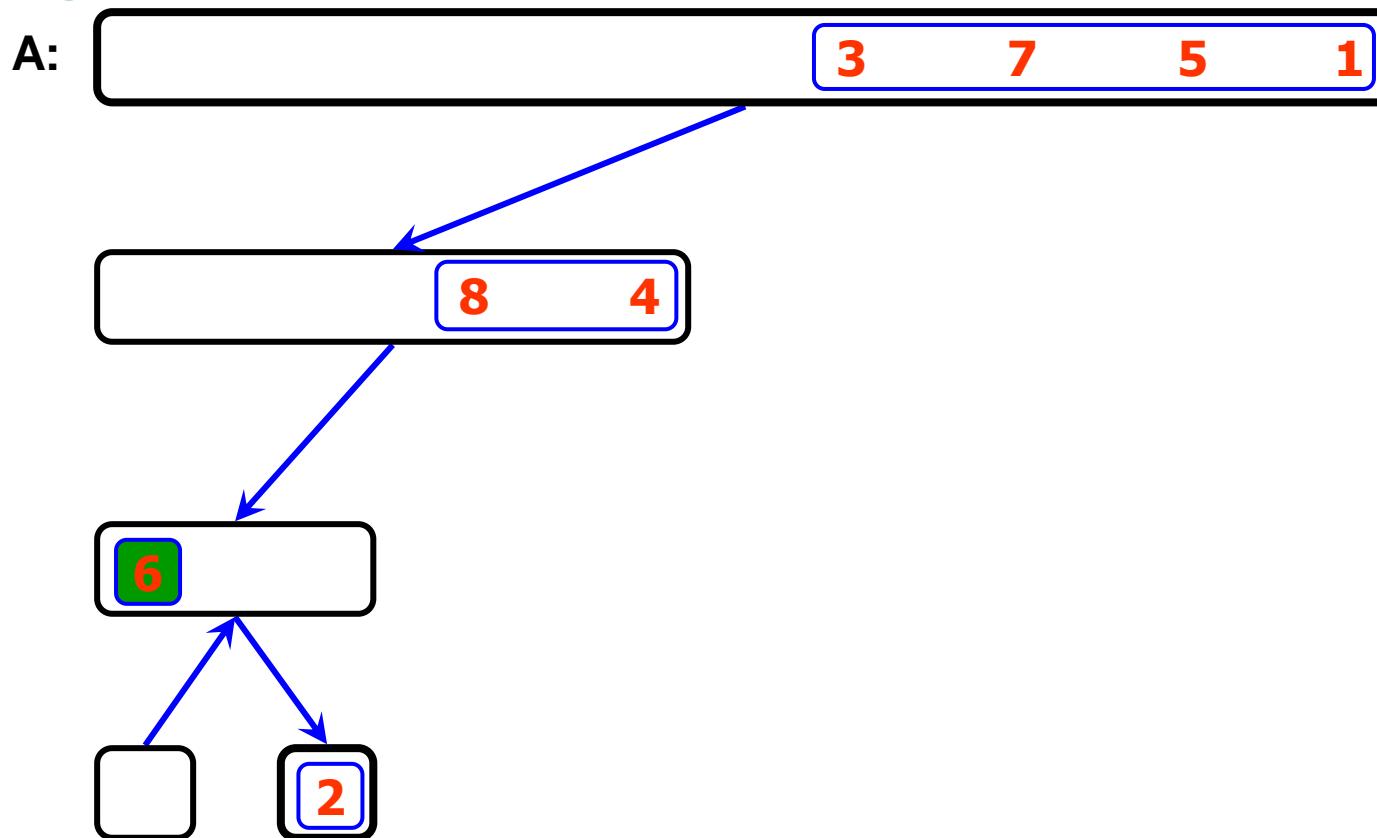
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 0), return



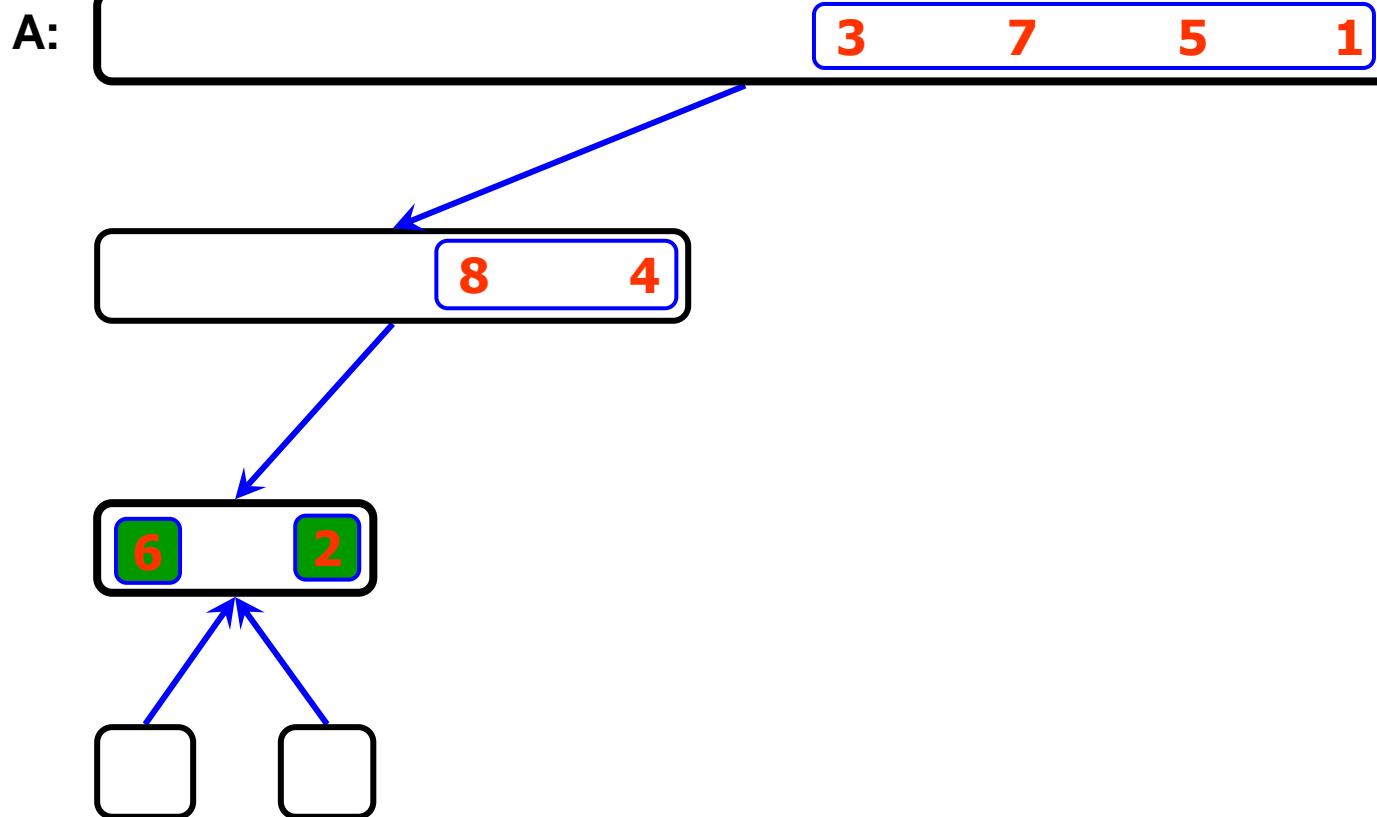
Merge-Sort(A, 0, 7)

Merge-Sort(A, 1, 1), base case



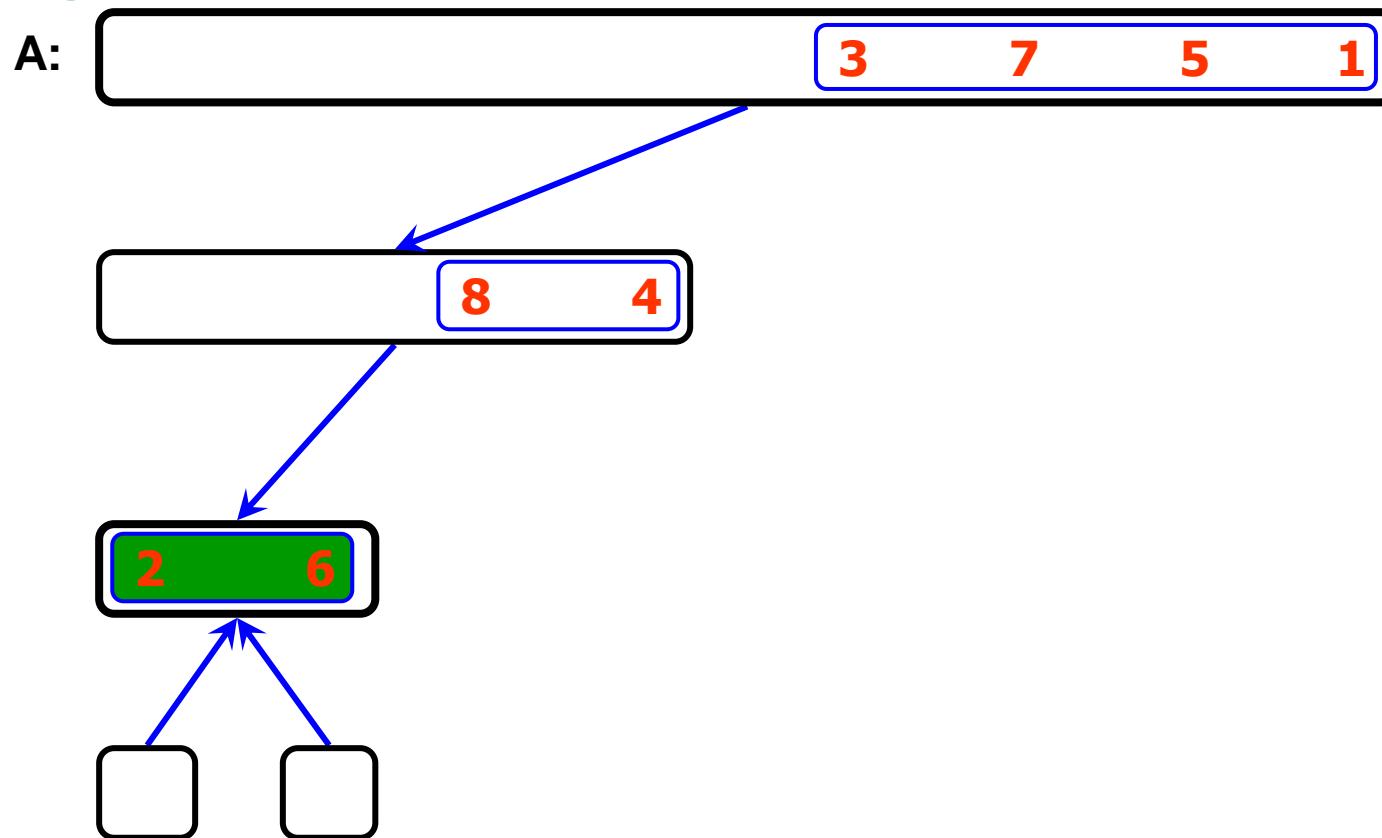
Merge-Sort(A, 0, 7)

Merge-Sort(A, 1, 1), return



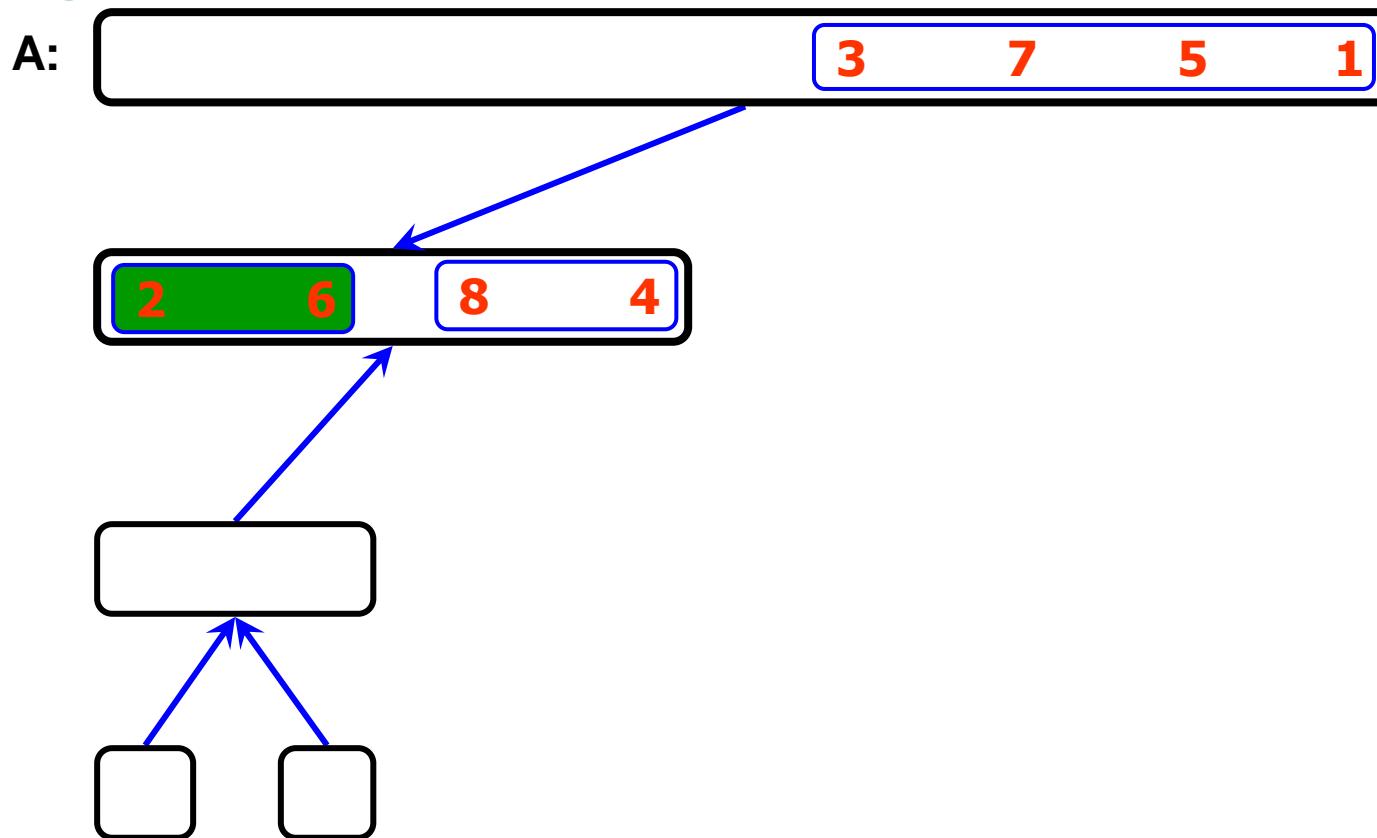
Merge-Sort(A, 0, 7)

Merge(A, 0, 0, 1)



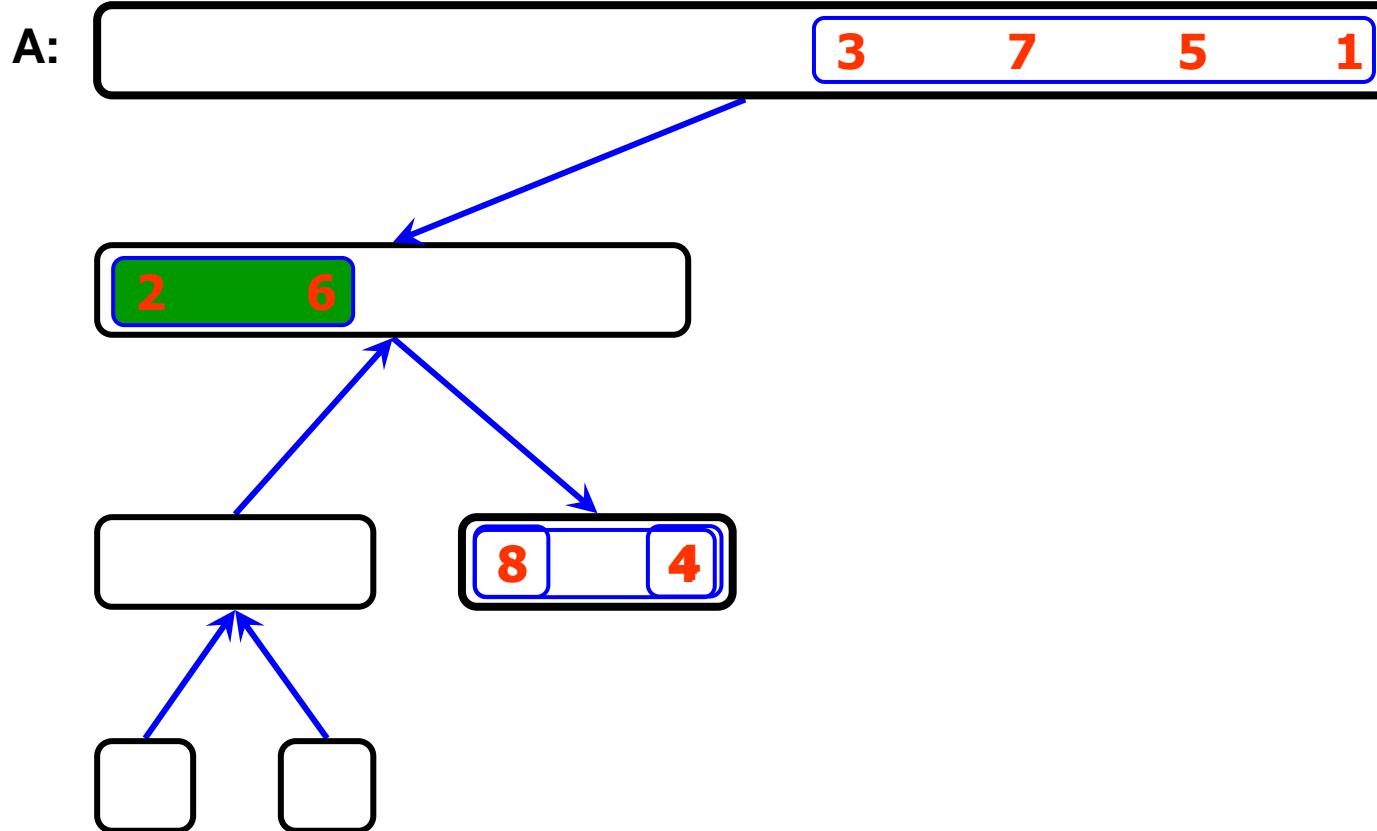
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 1), return



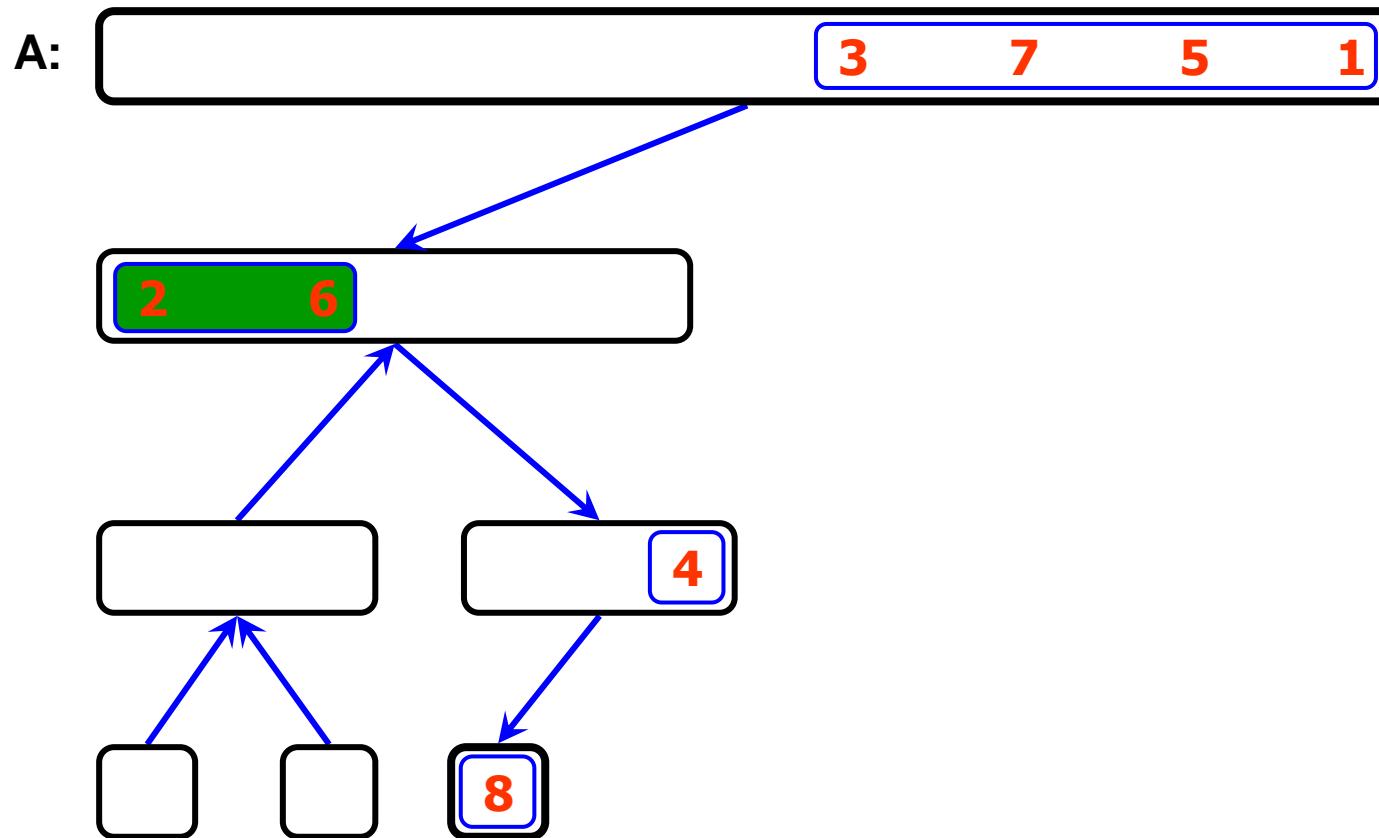
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 3), divide



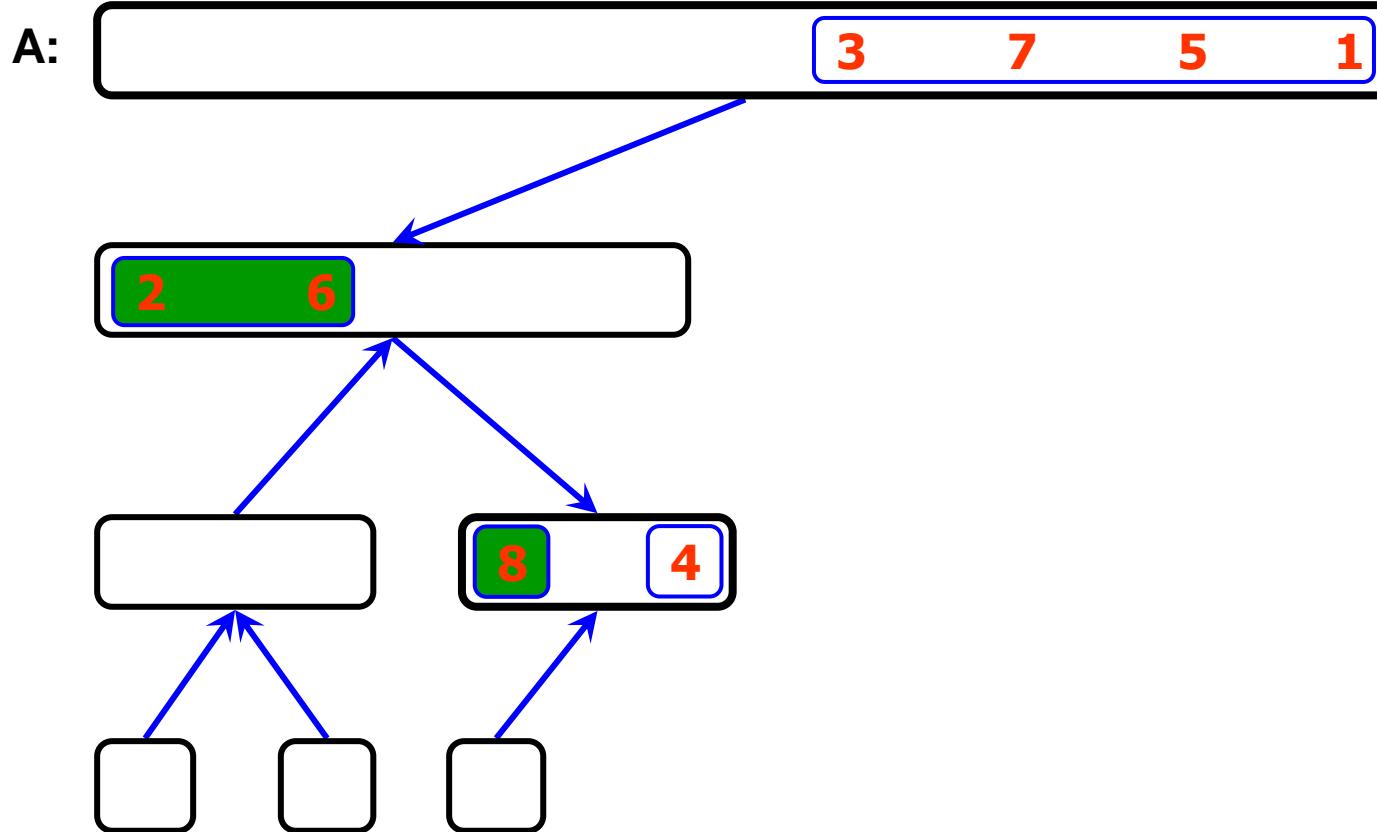
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 2), base case



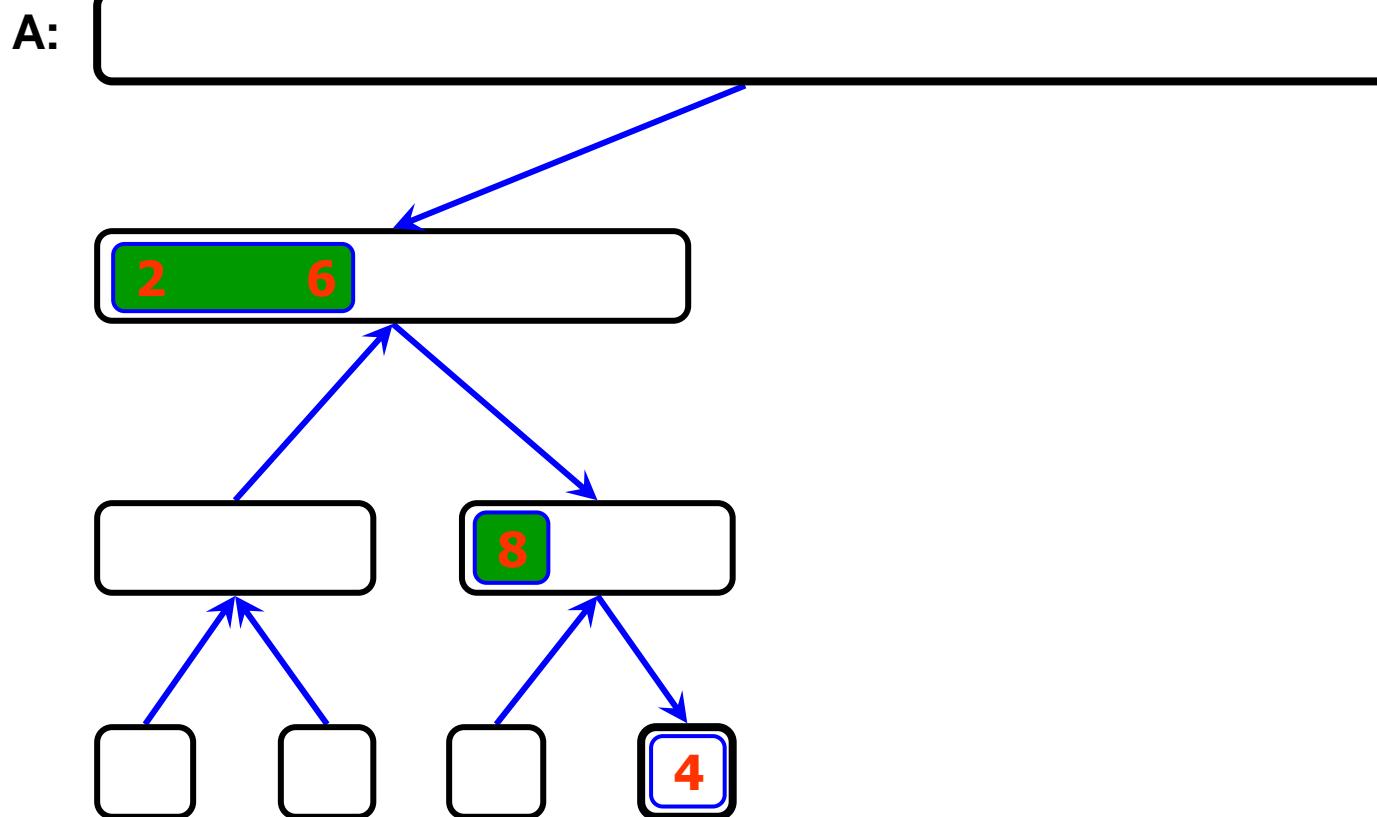
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 2), return



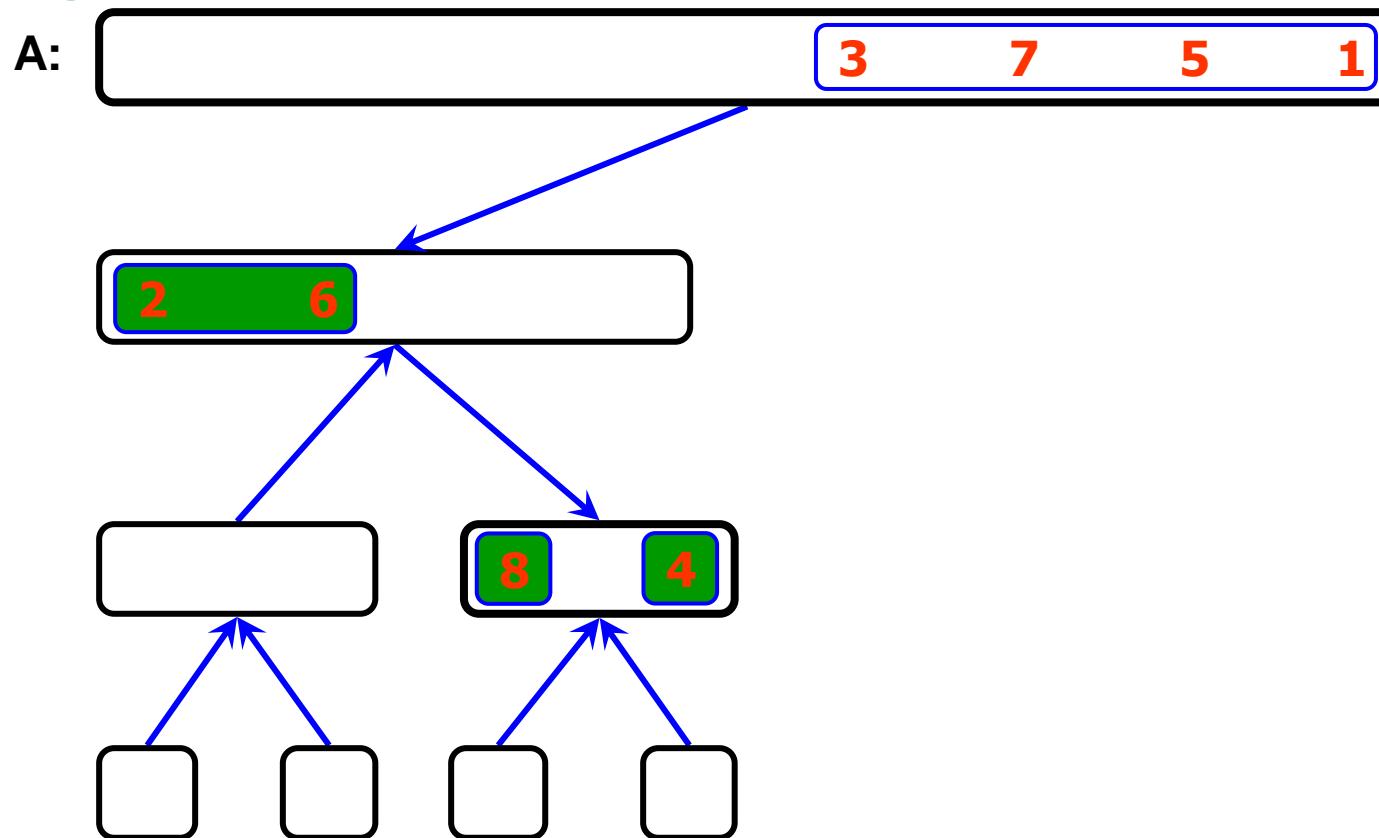
Merge-Sort(A, 0, 7)

Merge-Sort(A, 3, 3), base case



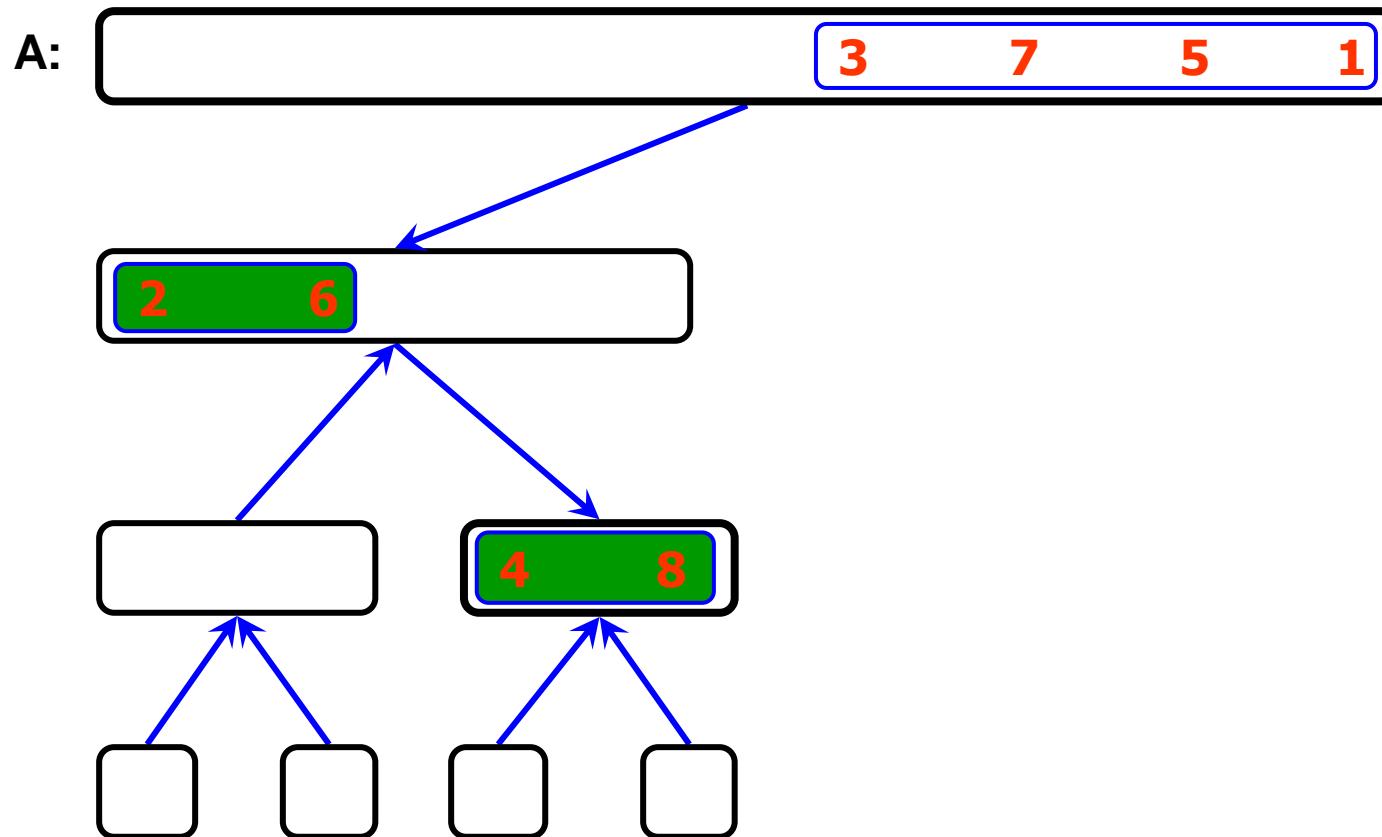
Merge-Sort(A, 0, 7)

Merge-Sort(A, 3, 3), return



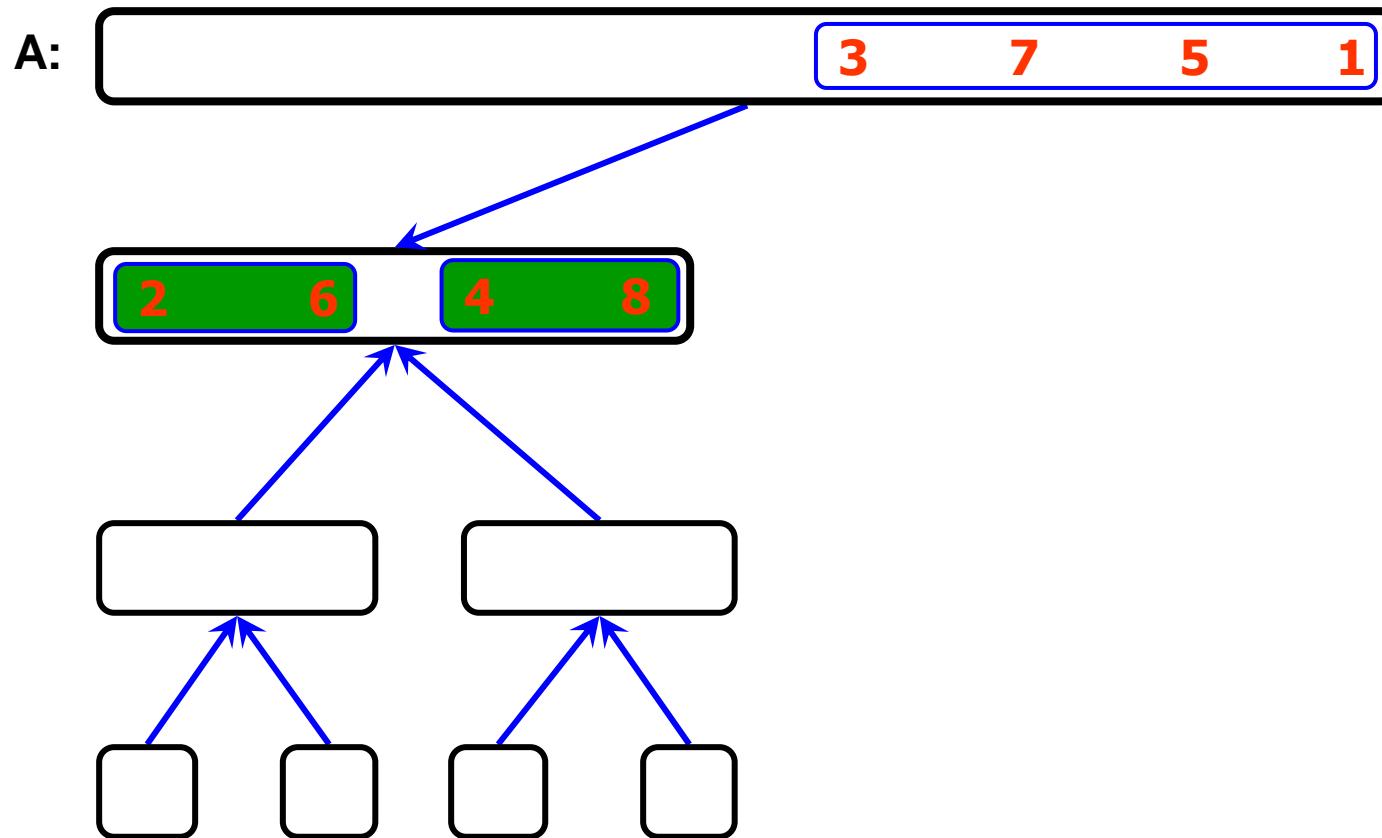
Merge-Sort(A, 0, 7)

Merge(A, 2, 2, 3)



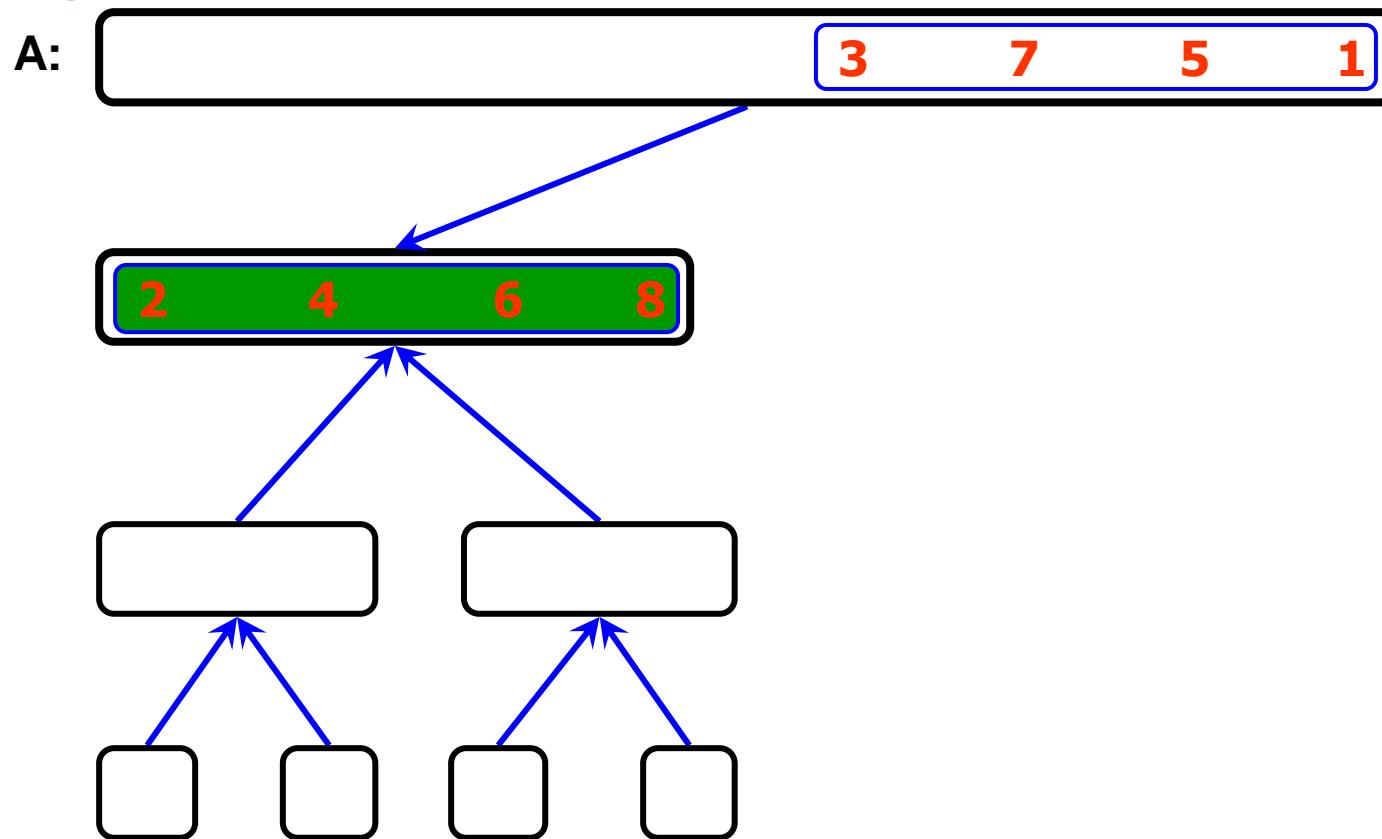
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 3), return



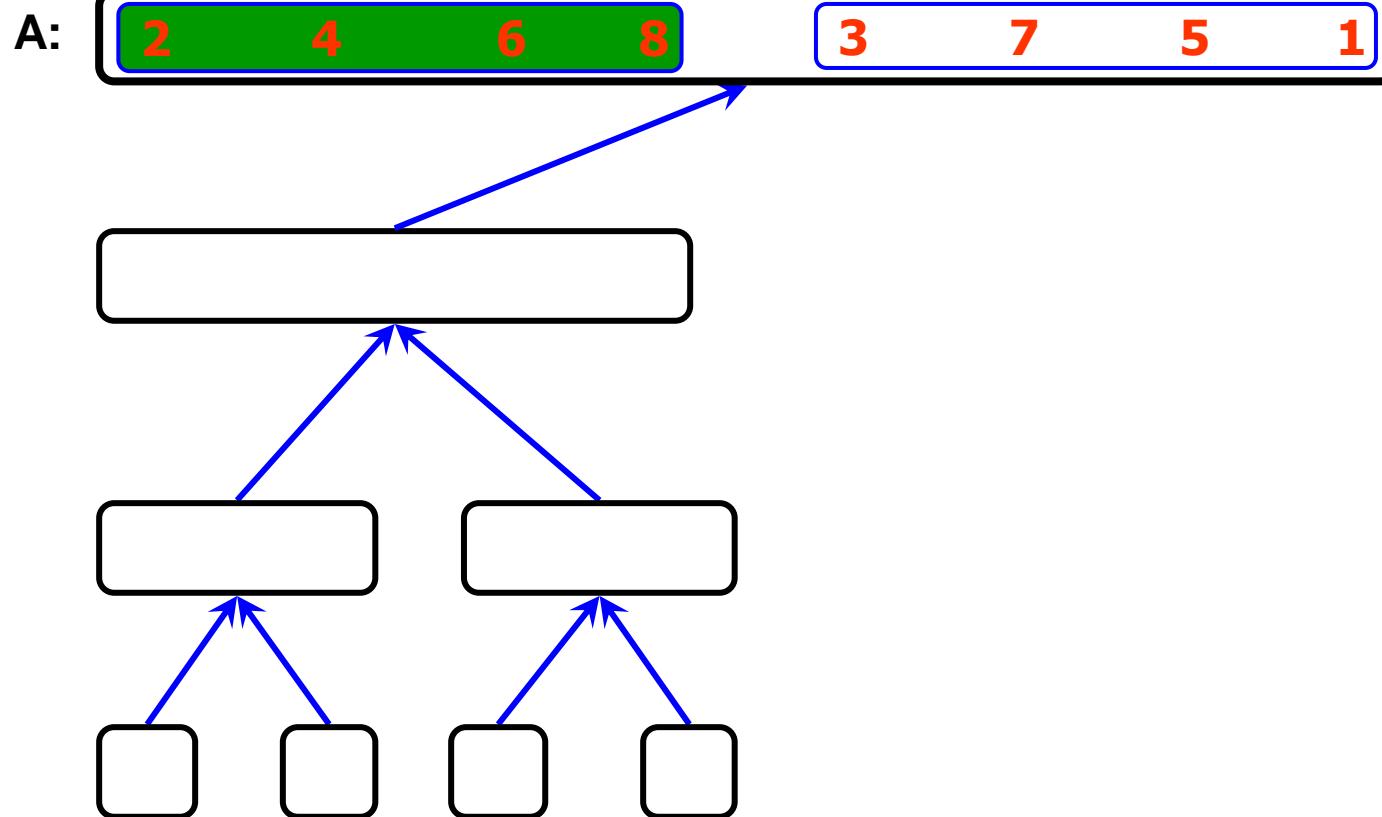
Merge-Sort(A, 0, 7)

Merge(A, 0, 1, 3)



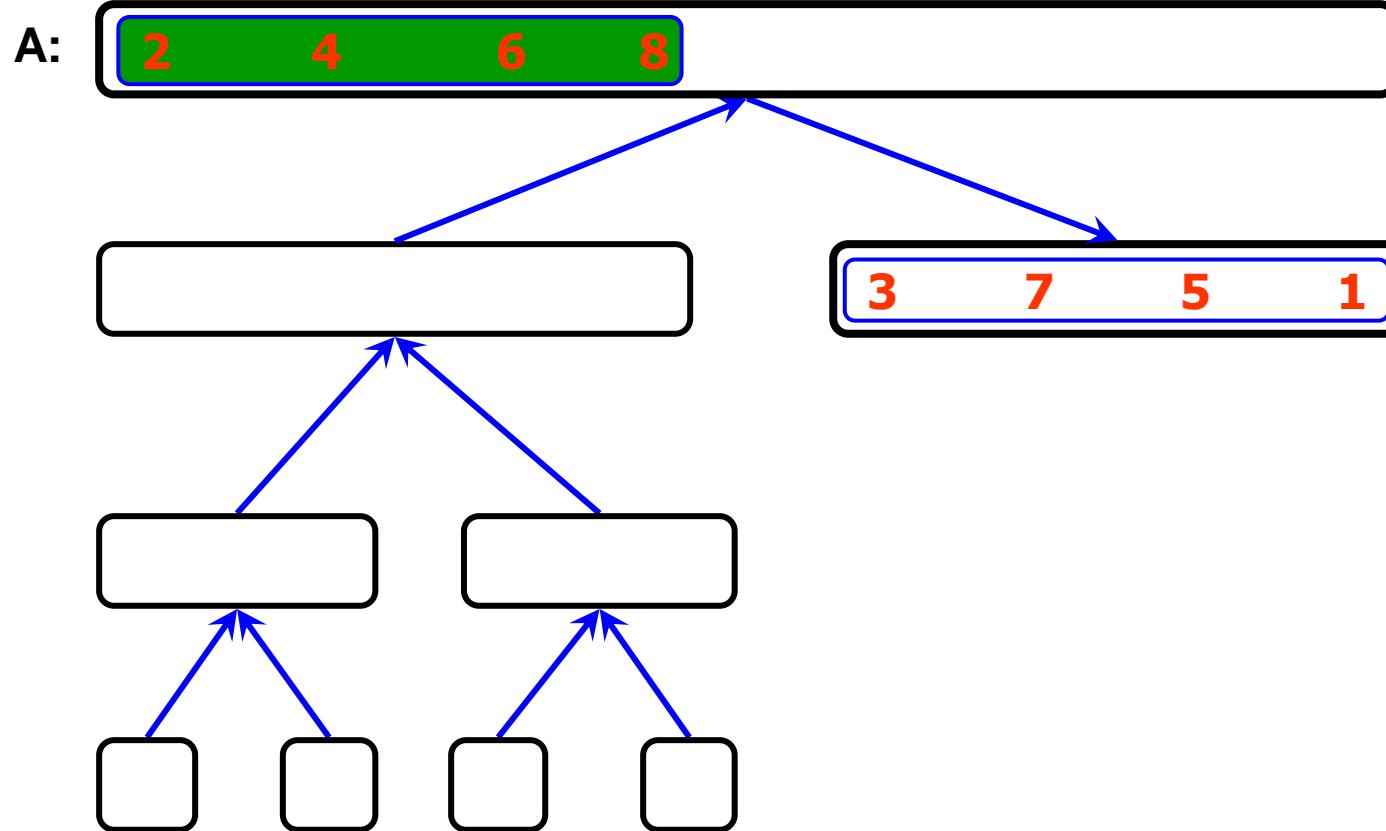
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 3), return



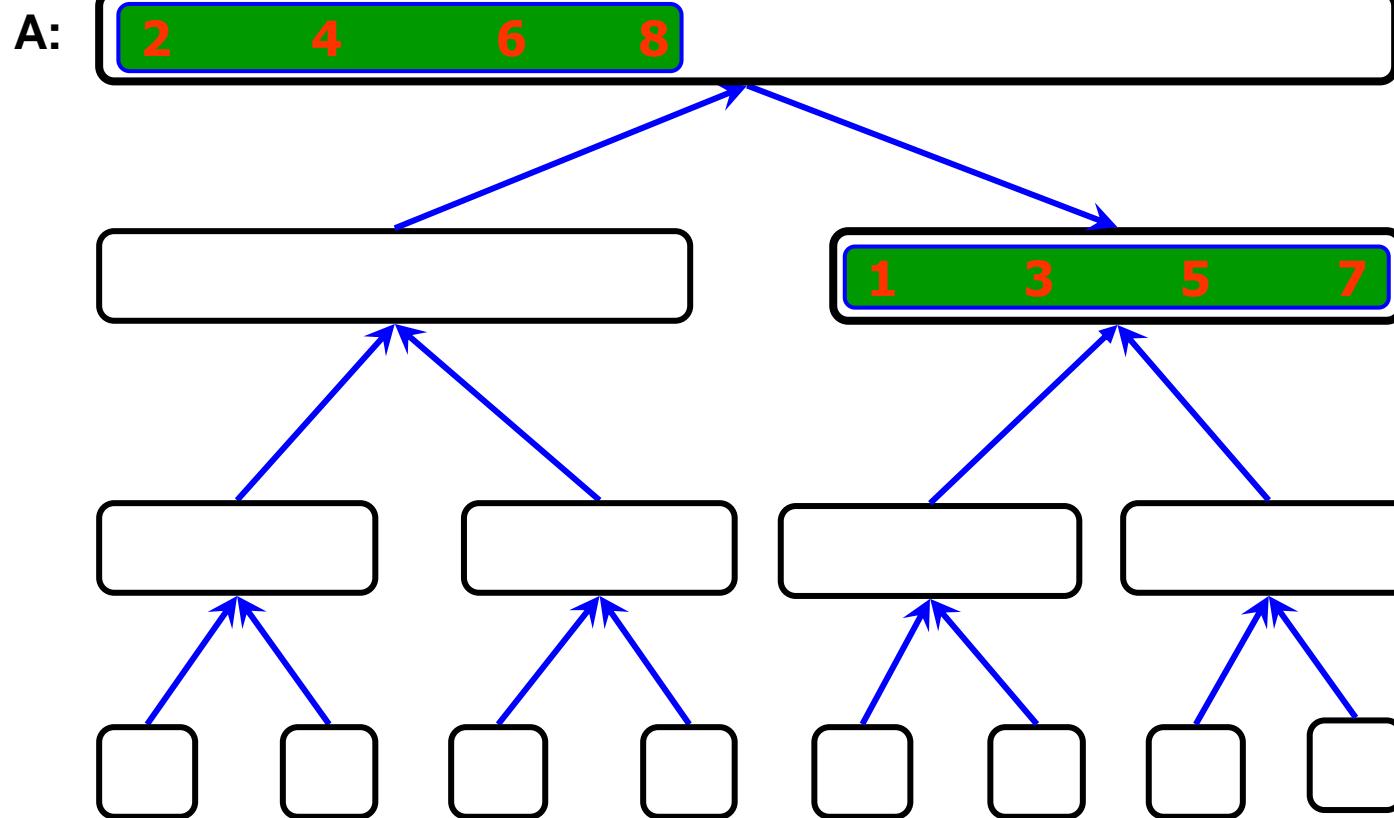
Merge-Sort(A, 0, 7)

Merge-Sort(A, 4, 7)



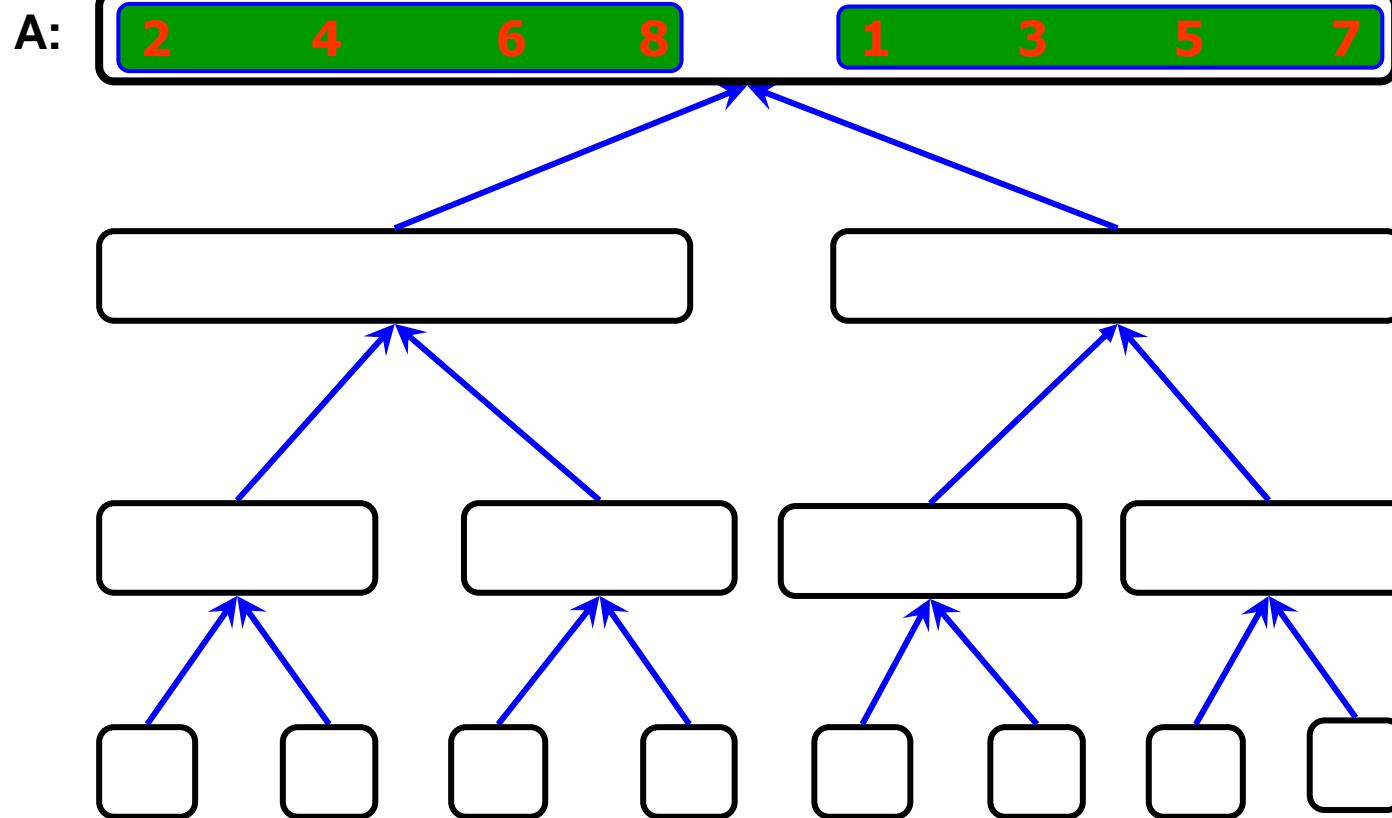
Merge-Sort(A, 0, 7)

Merge (A, 4, 5, 7)



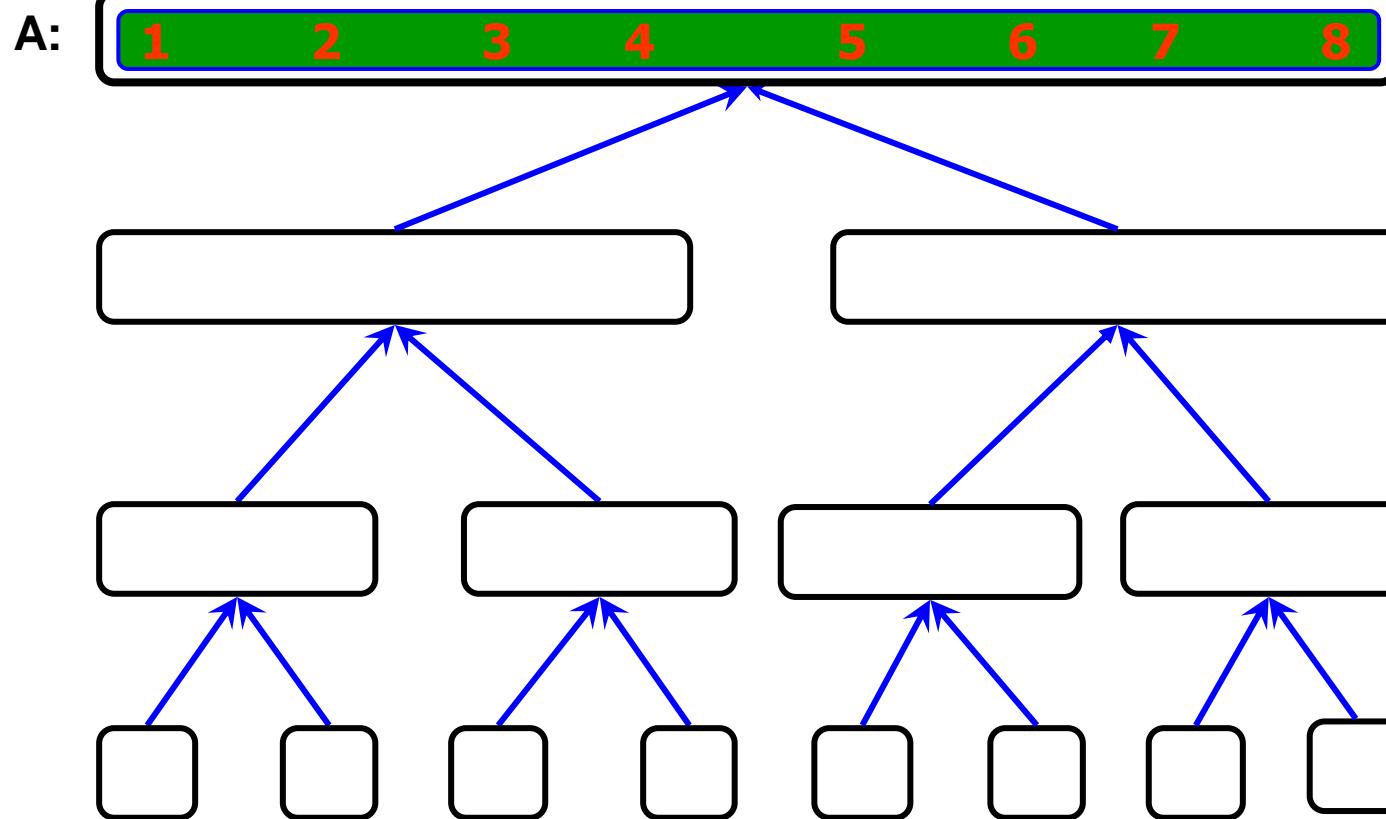
Merge-Sort(A, 0, 7)

Merge-Sort(A, 4, 7), return



Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 7), done!





MERGE SORT ALGORITHM AND CODING

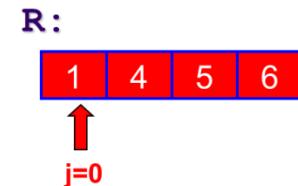
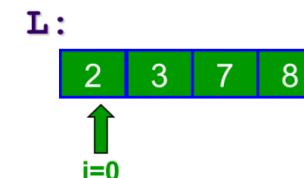
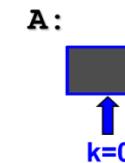
mergeSort

```
def mergeSort(arr):
    if len(arr) > 1:
        # Finding the mid of the array
        mid = len(arr)//2
        # Dividing the array elements
        L = arr[:mid]
        R = arr[mid:]
        # Sorting the first half
        mergeSort(L)
        # Sorting the second half
        mergeSort(R)
```

demo: MergeSort.py

Merge(A, left, middle, right)

```
1.  $n_1 \leftarrow \text{middle} - \text{left} + 1$ 
2.  $n_2 \leftarrow \text{right} - \text{middle}$ 
3. create array L[ $n_1$ ], R[ $n_2$ ]
4. for  $i \leftarrow 0$  to  $n_1-1$  do  $L[i] \leftarrow A[\text{left}+i]$ 
5. for  $j \leftarrow 0$  to  $n_2-1$  do  $R[j] \leftarrow A[\text{middle}+j+1]$ 
6.  $k \leftarrow \text{left}$ 
7.  $i \leftarrow j \leftarrow 0$ 
8. while  $i < n_1 \& j < n_2$ 
9.     if  $L[i] < R[j]$ 
10.         $A[k++] \leftarrow L[i++]$ 
11.    else
12.         $A[k++] \leftarrow R[j++]$ 
13. while  $i < n_1$ 
14.      $A[k++] \leftarrow L[i++]$ 
15. while  $j < n_2$ 
16.      $A[k++] \leftarrow R[j++]$ 
```



$$n = n_1 + n_2$$

Space: n

Time : cn for some constant c

A:



merge

```
# Copy data to temp arrays L[] and R[]
while i < len(L) and j < len(R):
    if L[i] < R[j]:
        arr[k] = L[i]
        i += 1
    else:
        arr[k] = R[j]
        j += 1
    k += 1

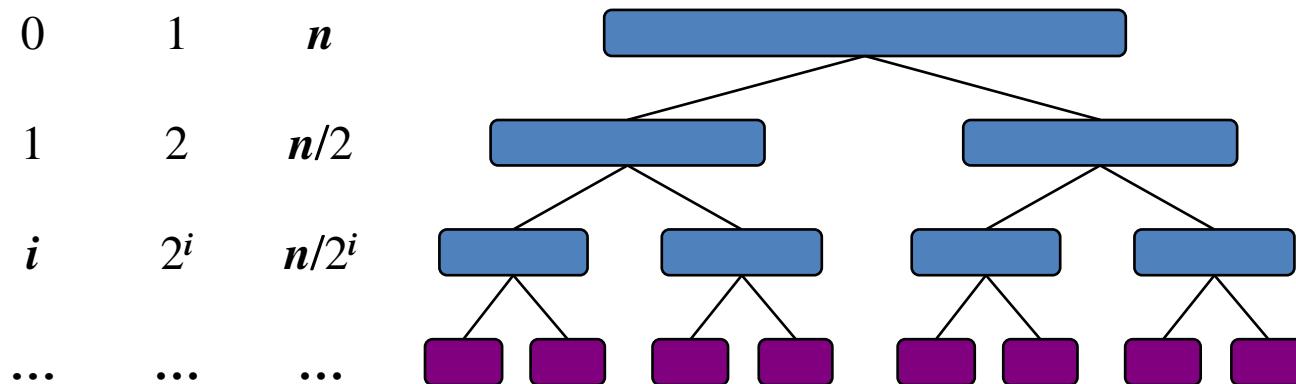
# Checking if any element was left
while i < len(L):
    arr[k] = L[i]
    i += 1
    k += 1

while j < len(R):
    arr[k] = R[j]
    j += 1
    k += 1
```

Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
 - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
 - we partition and merge 2^i sequences of size $n/2^i$
 - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth #seqs size



QUICK SORT

Quick sort

- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say **pivot**.
- Based on *pivot* the partition is made and another array holds values greater than the pivot value.

Quick Sort

- **Divide:**
 - Pick any element **p** as the **pivot**, e.g, the first element
 - **Partition** the remaining elements into
 - FirstPart**, which contains all elements $< p$
 - SecondPart**, which contains all elements $\geq p$
- **Recursively sort** the **FirstPart** and **SecondPart**
- **Combine:** no work is necessary since sorting is done in place



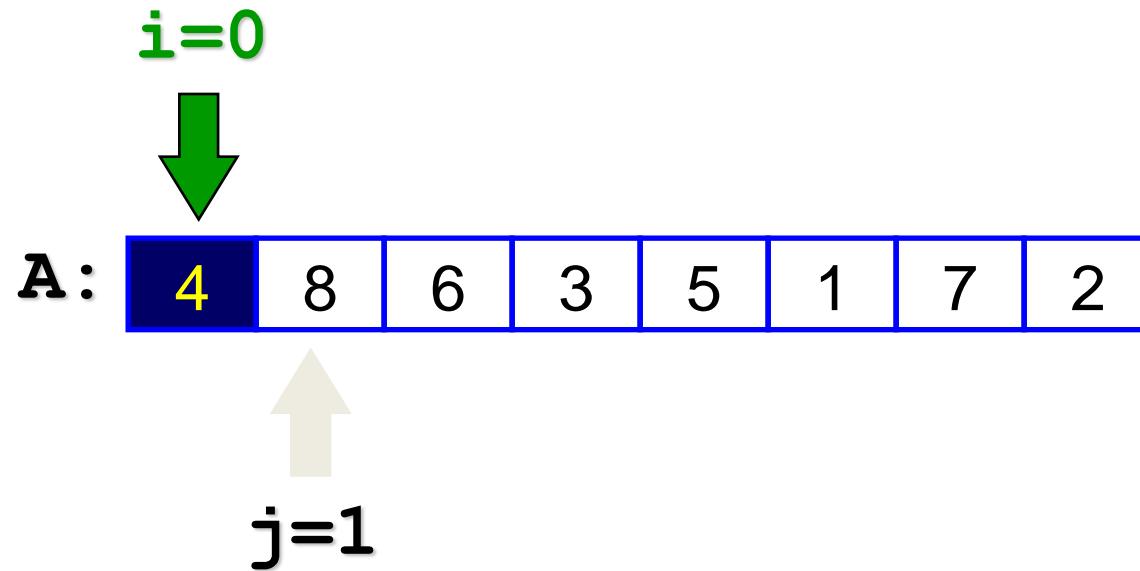
PARTITION IN ARRAY IN QUICK SORT

Partition Example

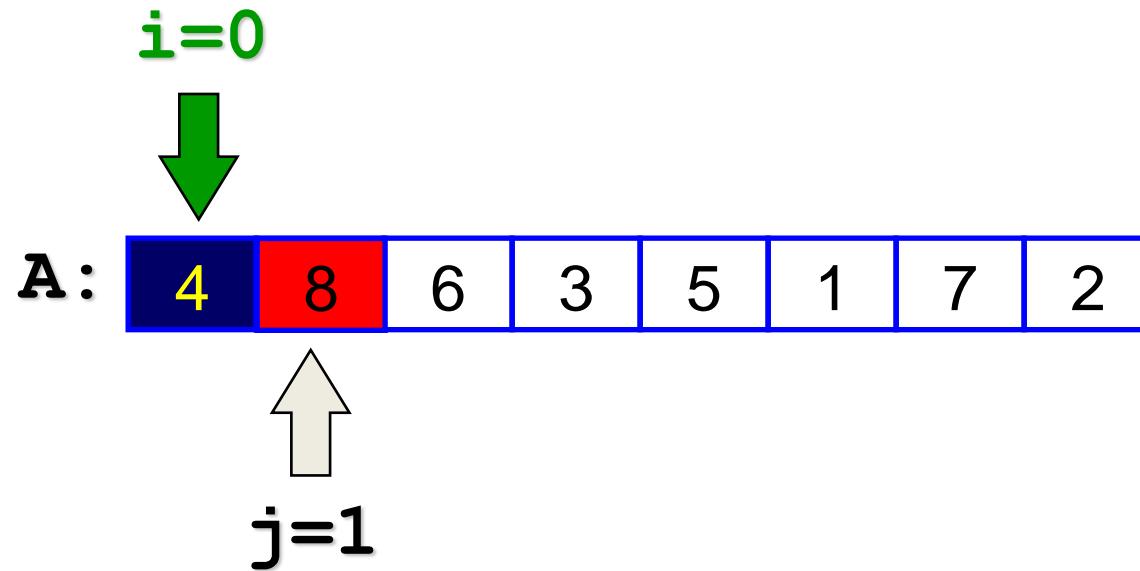
A:

4	8	6	3	5	1	7	2
---	---	---	---	---	---	---	---

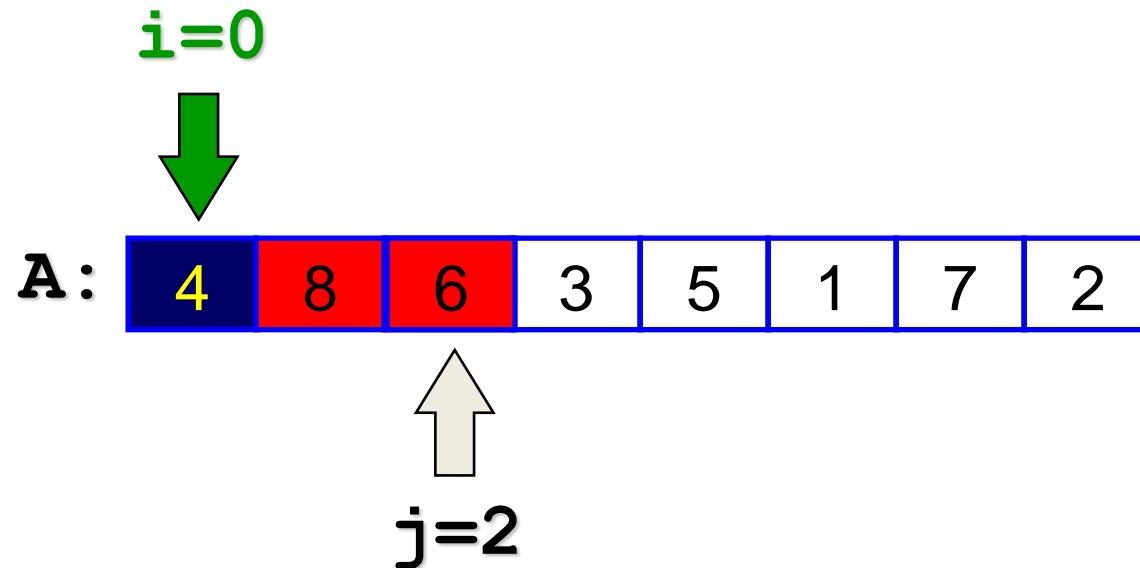
Partition Example



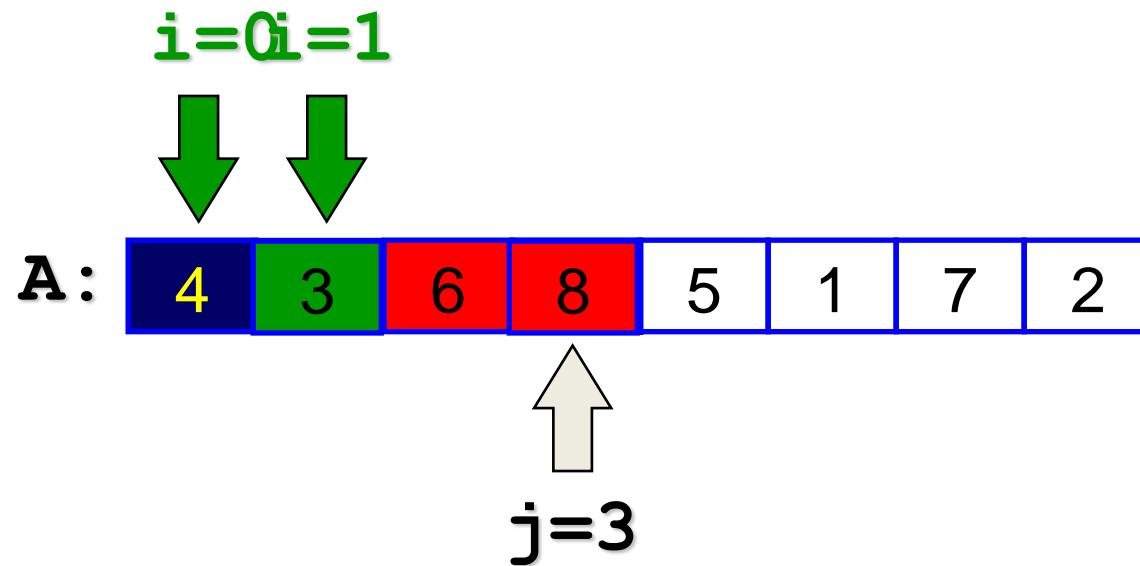
Partition Example



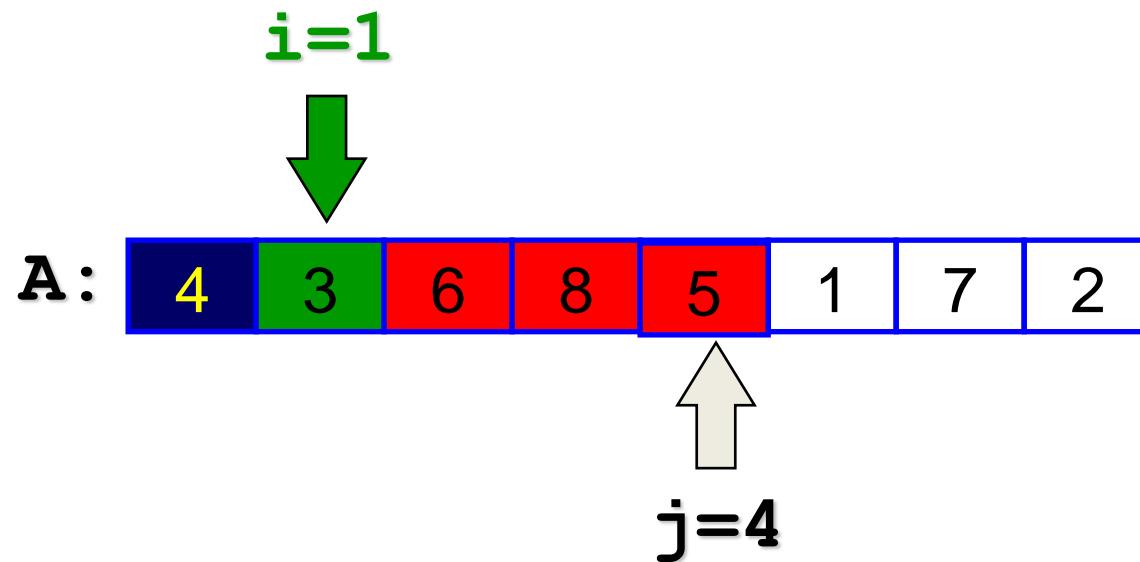
Partition Example



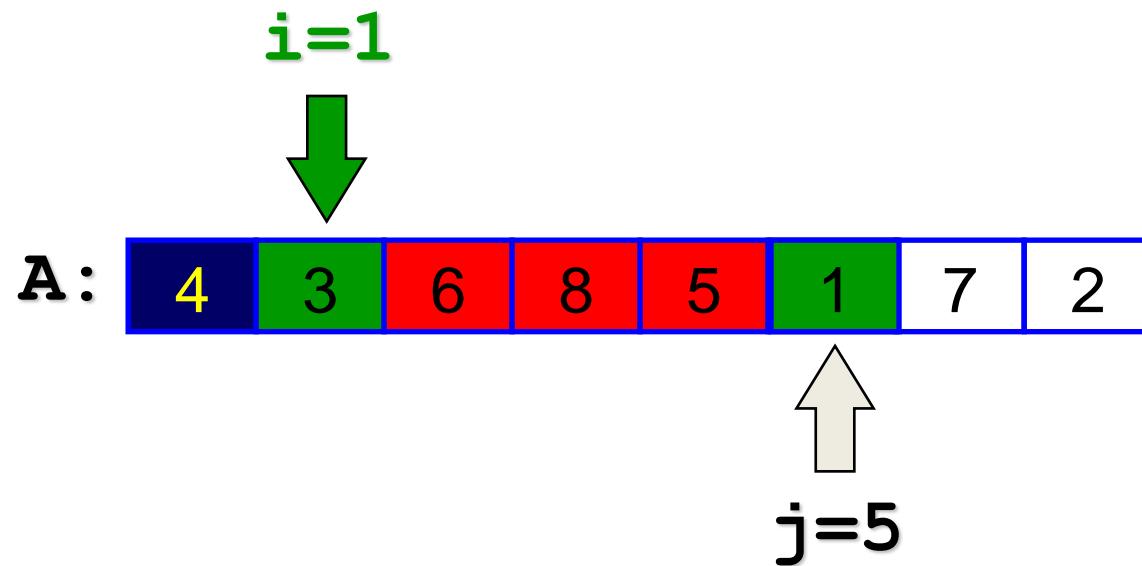
Partition Example



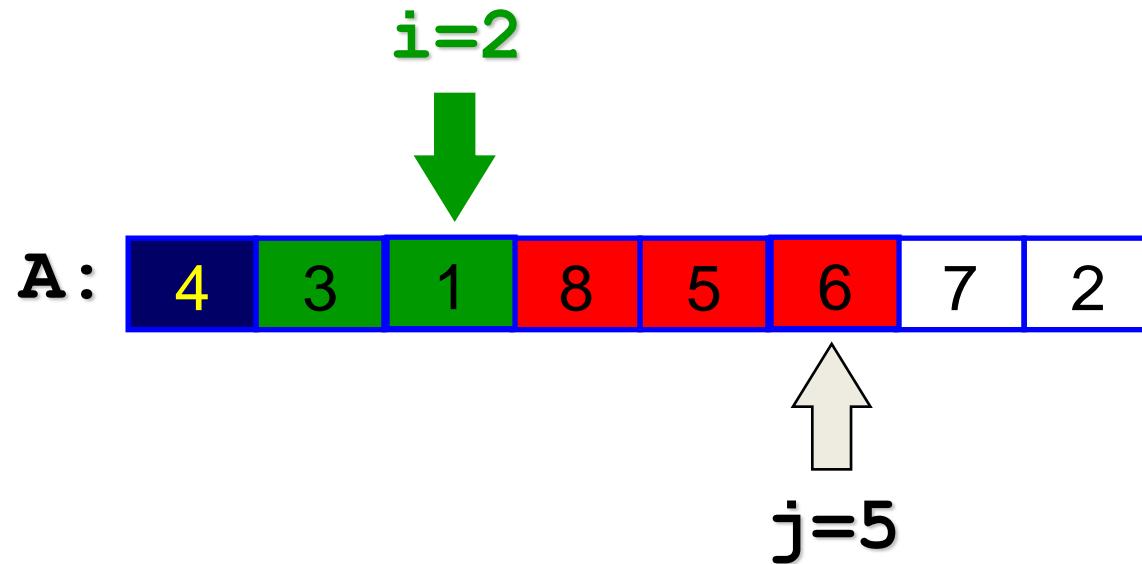
Partition Example



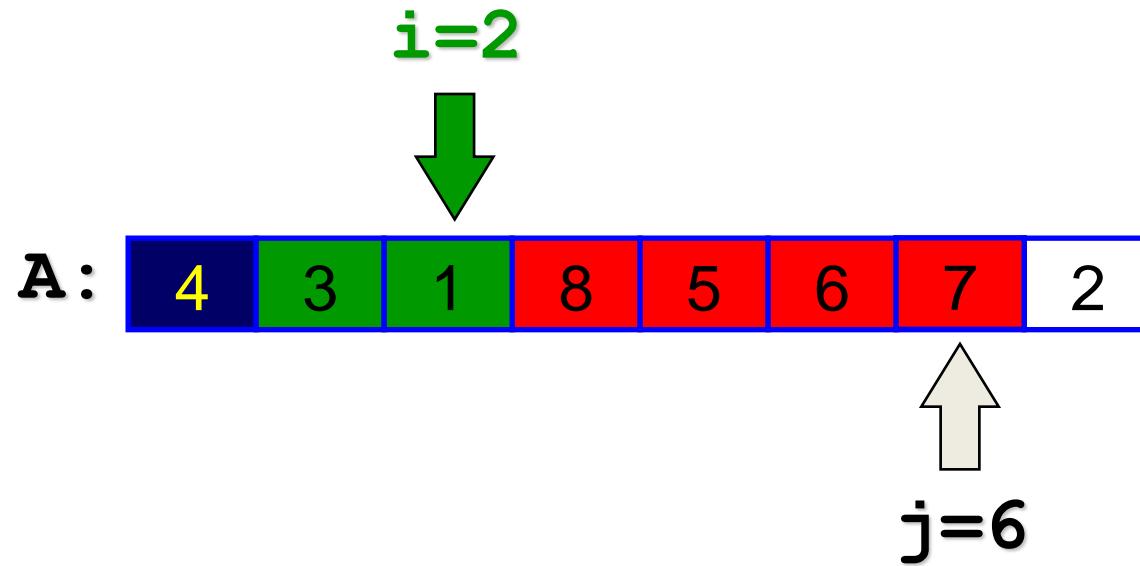
Partition Example



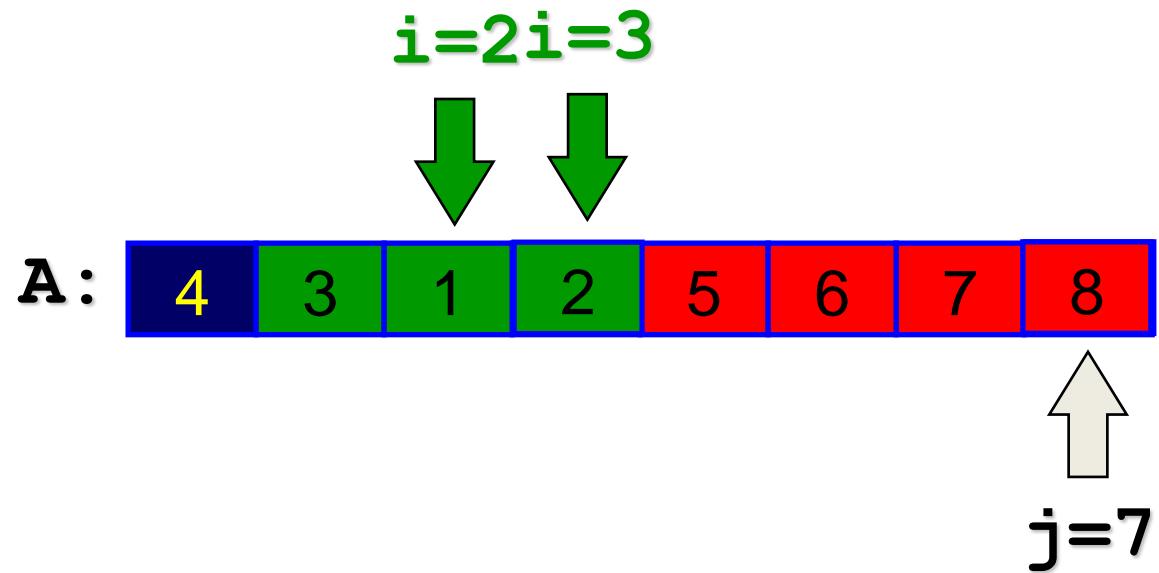
Partition Example



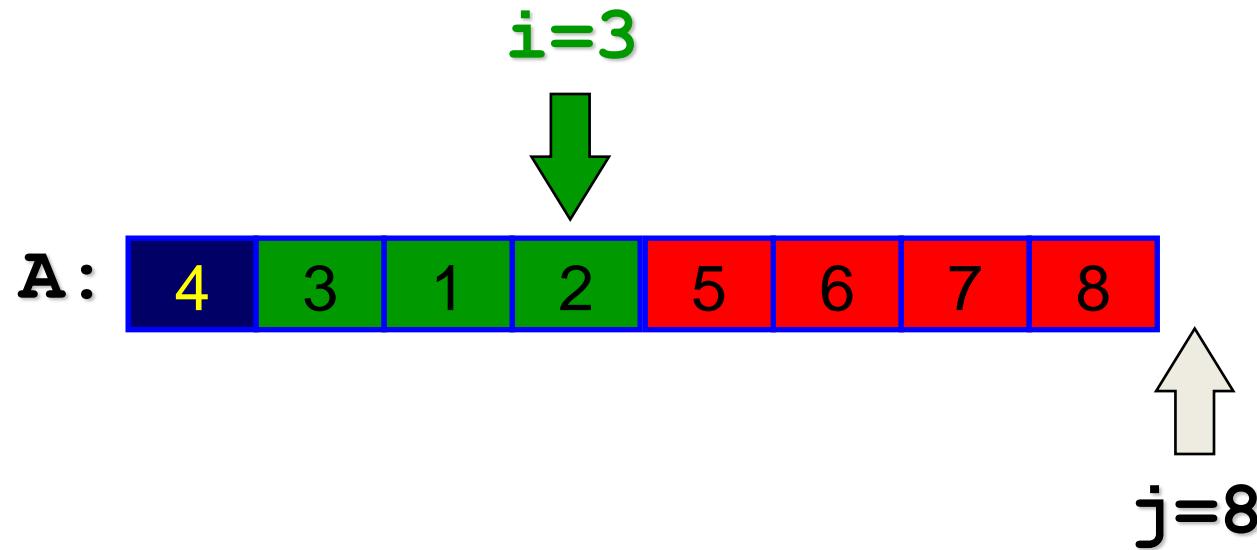
Partition Example



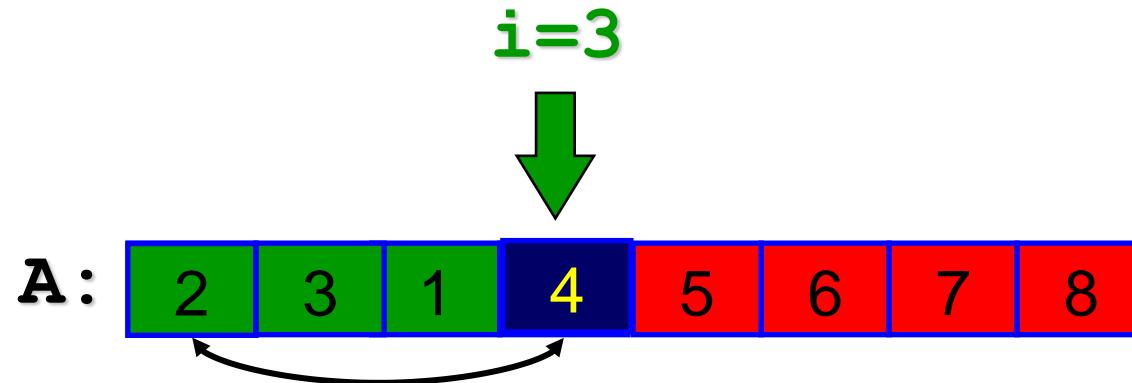
Partition Example



Partition Example

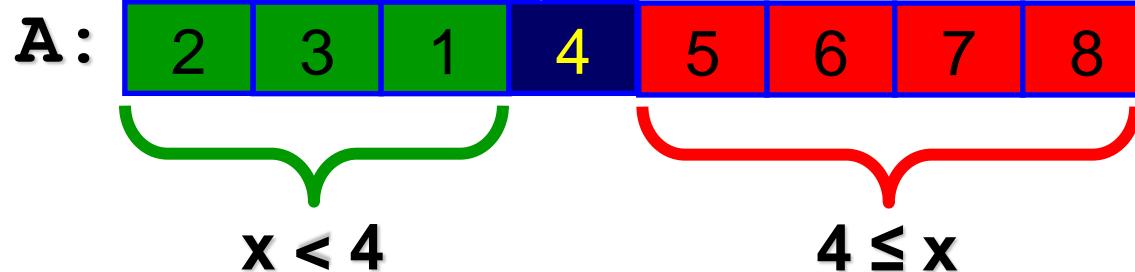


Partition Example



Partition Example

pivot in
correct position



Algorithm of Partition

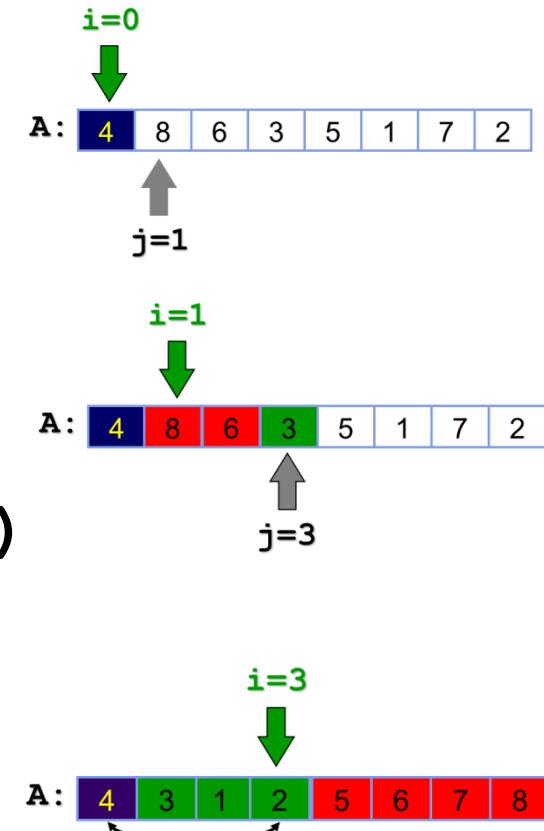
Partition(A, left, right)

```
1. x ← A[left]
2. i ← left
3. for j ← left+1 to right
4.     if A[j] < x then
5.         i ← i + 1
6.         swap(A[i], A[j])
7.     end if
8. end for j
9. swap(A[i], A[left])
10. return i
```

n = right – left +1

Time: cn for some constant c

Space: constant





QUICK SORT ILLUSTRATION

Quick-Sort(A, 0, 7)

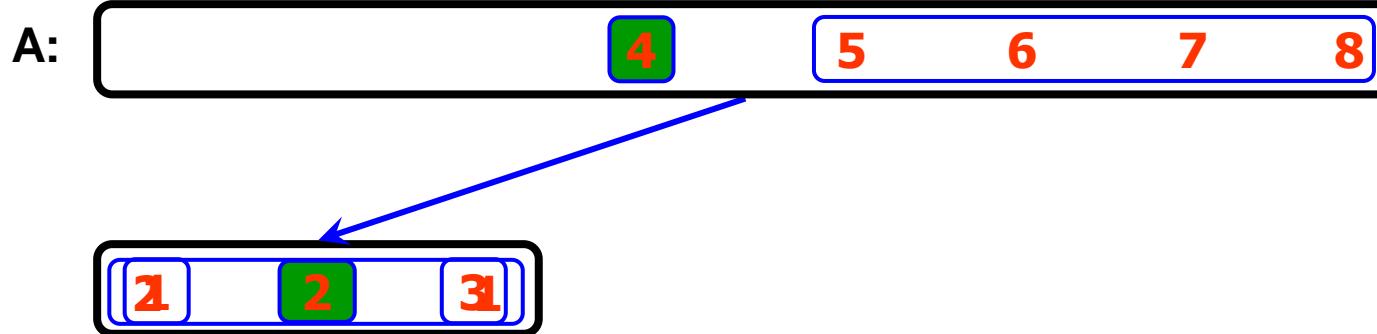
Partition

A:



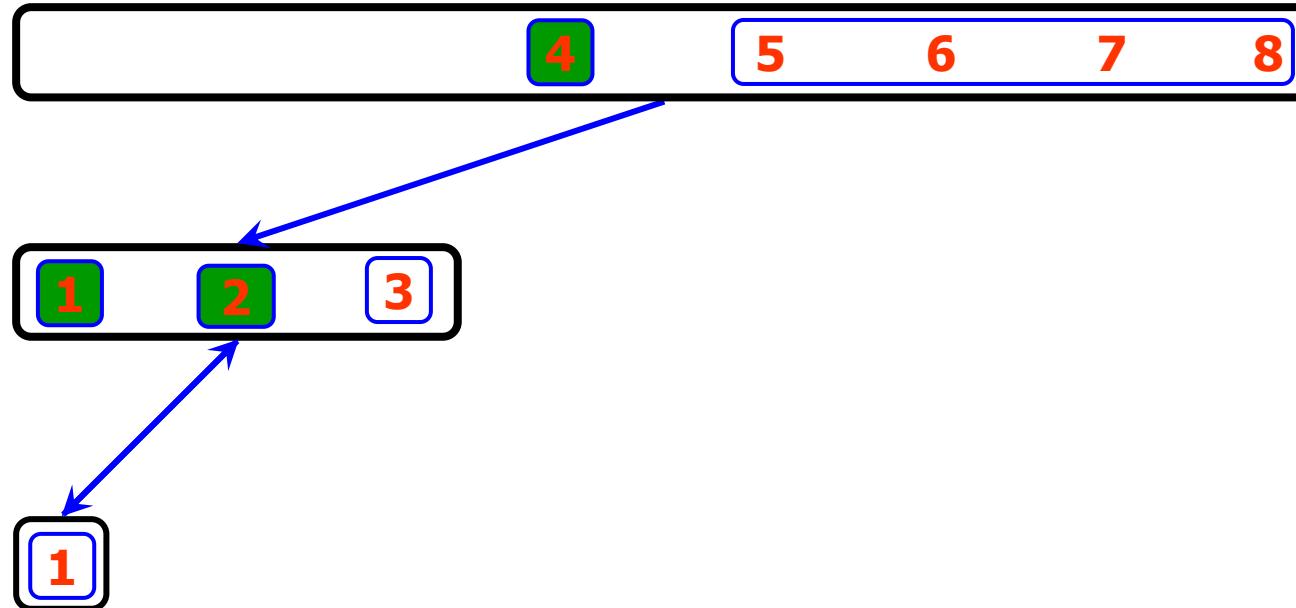
Quick-Sort(A, 0, 7)

Quick-Sort(A, 0, 2), partition



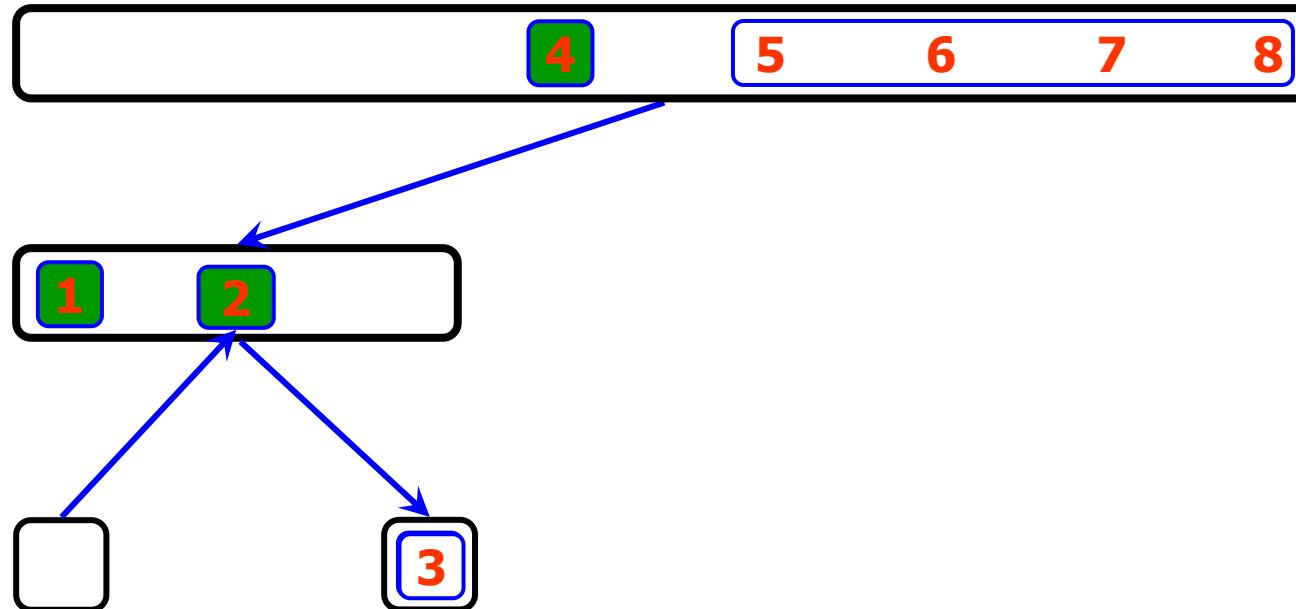
Quick-Sort(A, 0, 7)

Quick-Sort(A, 0, 0), ~~best~~ ~~worst~~ case



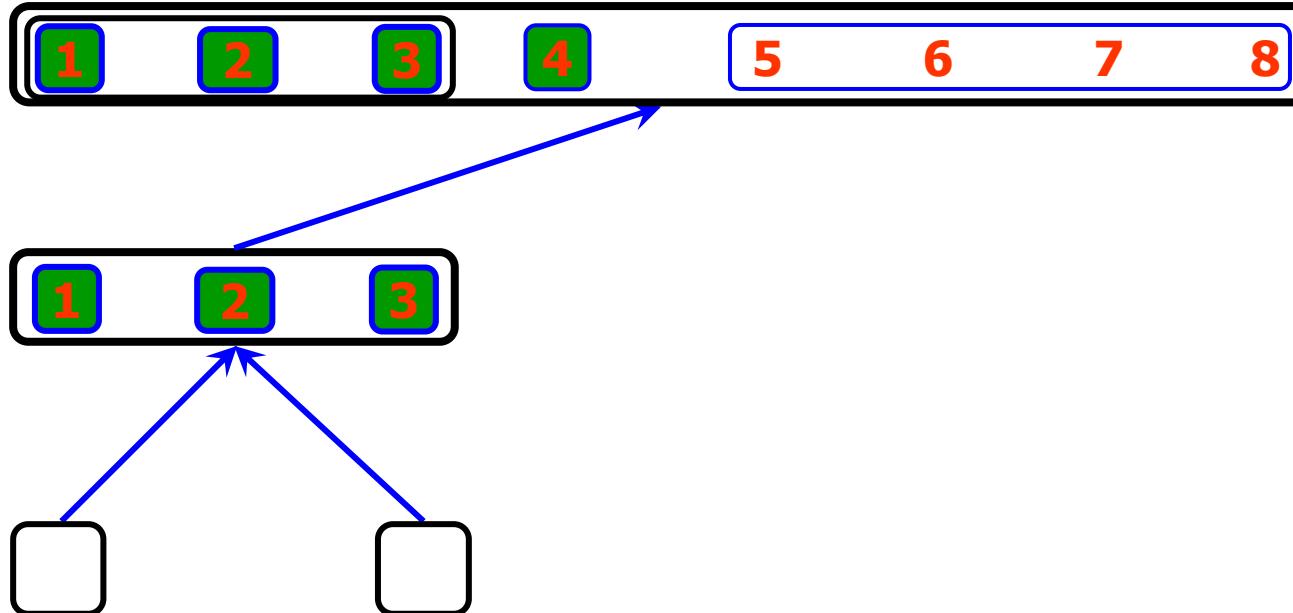
Quick-Sort(A, 0, 7)

Quick-Sort(A, 1, 1), base case



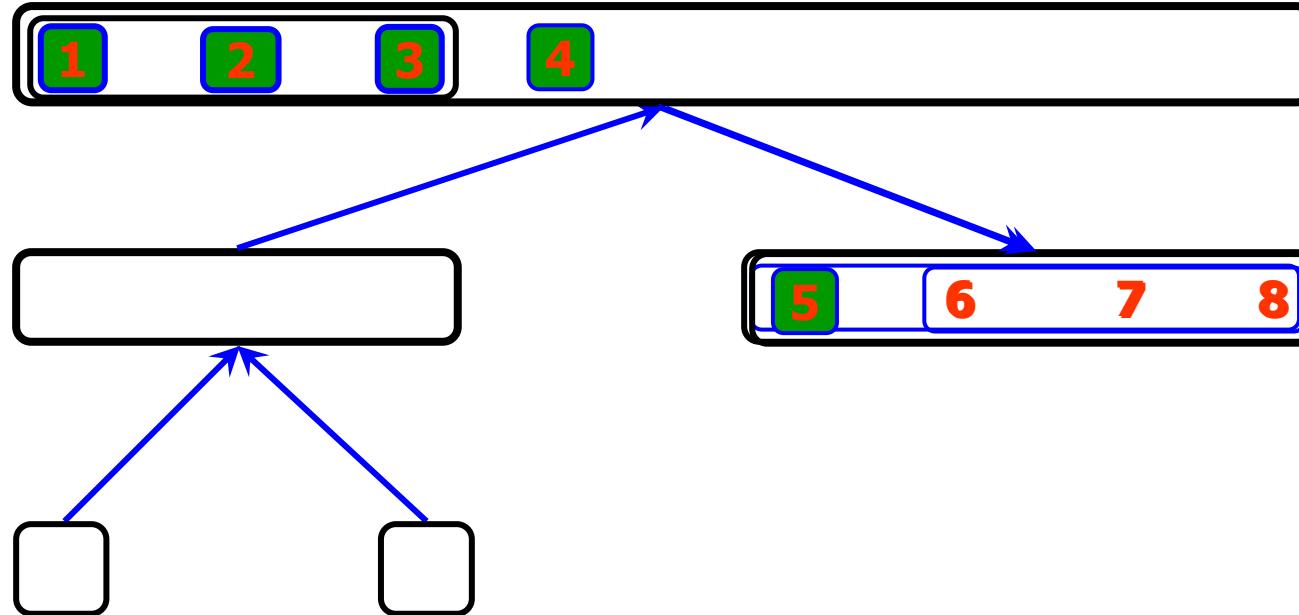
Quick-Sort(A, 0, 7)

Quick-Sort(A, 0, 2), return



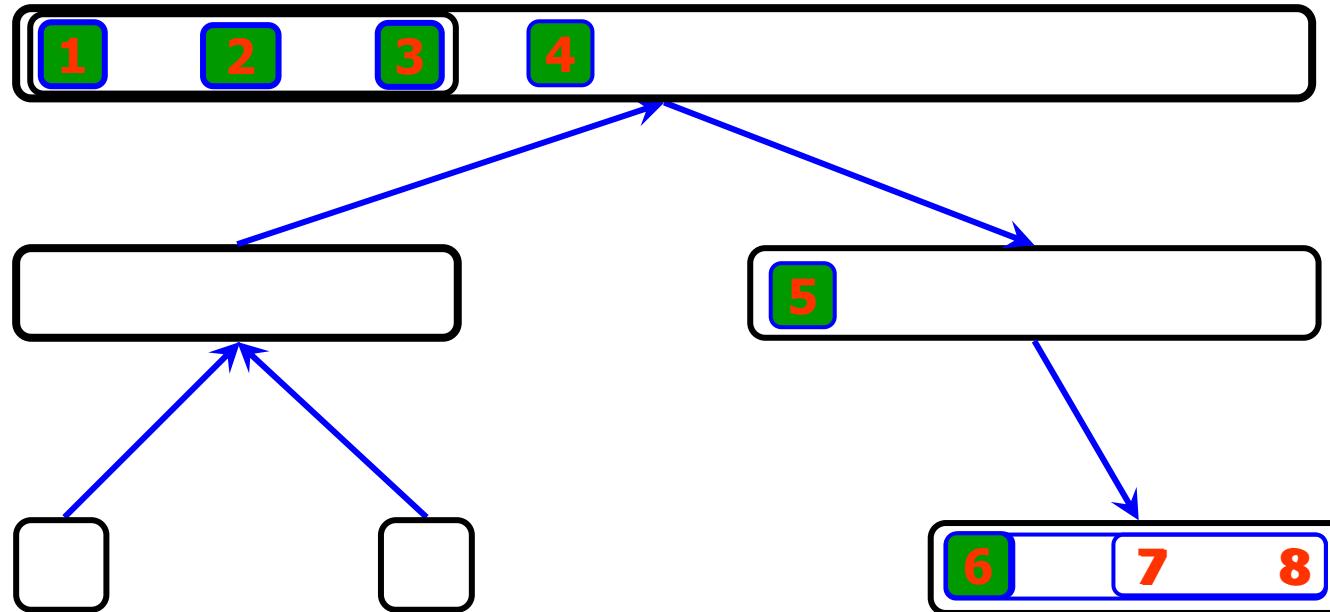
Quick-Sort(A, 0, 7)

Quick-Sort(A, 4, 7), partition



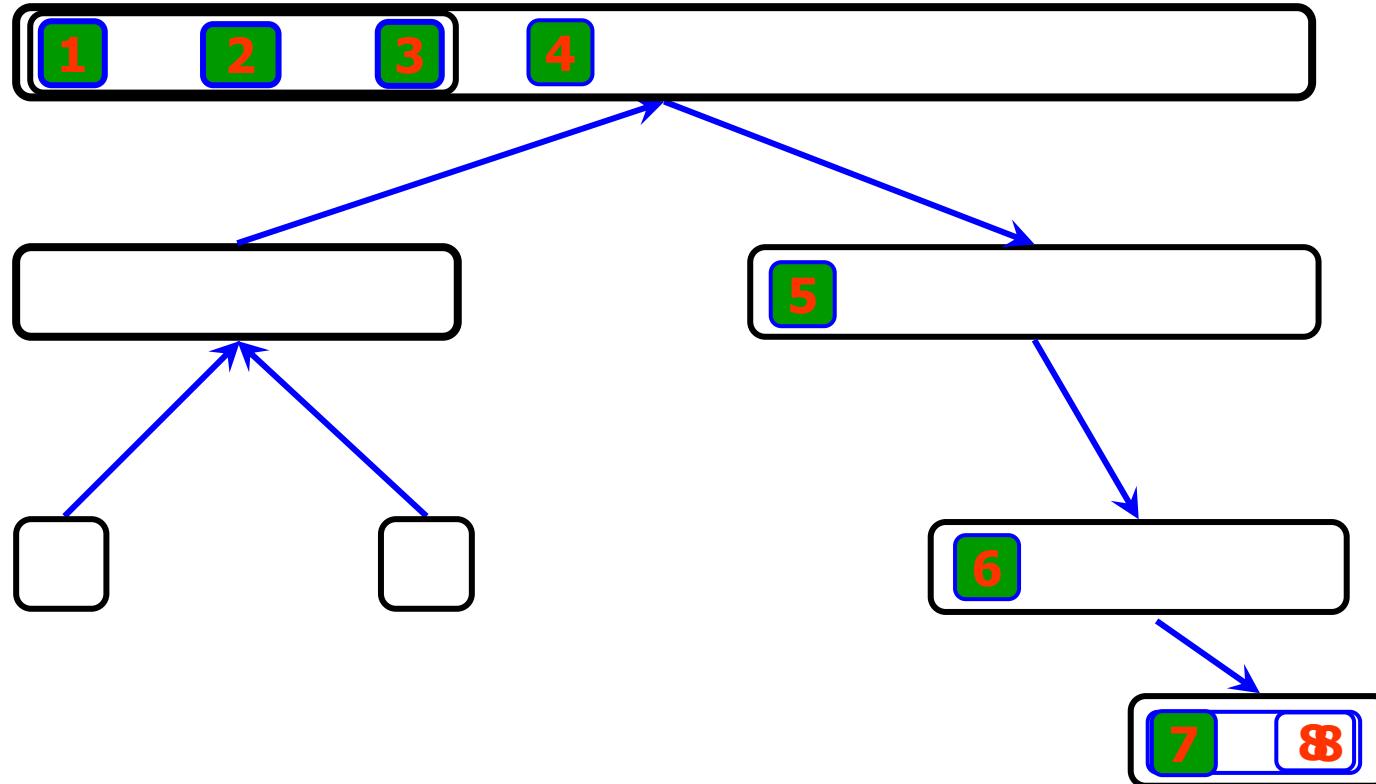
Quick-Sort(A, 0, 7)

Quick-Sort(A, 5, 7), partition



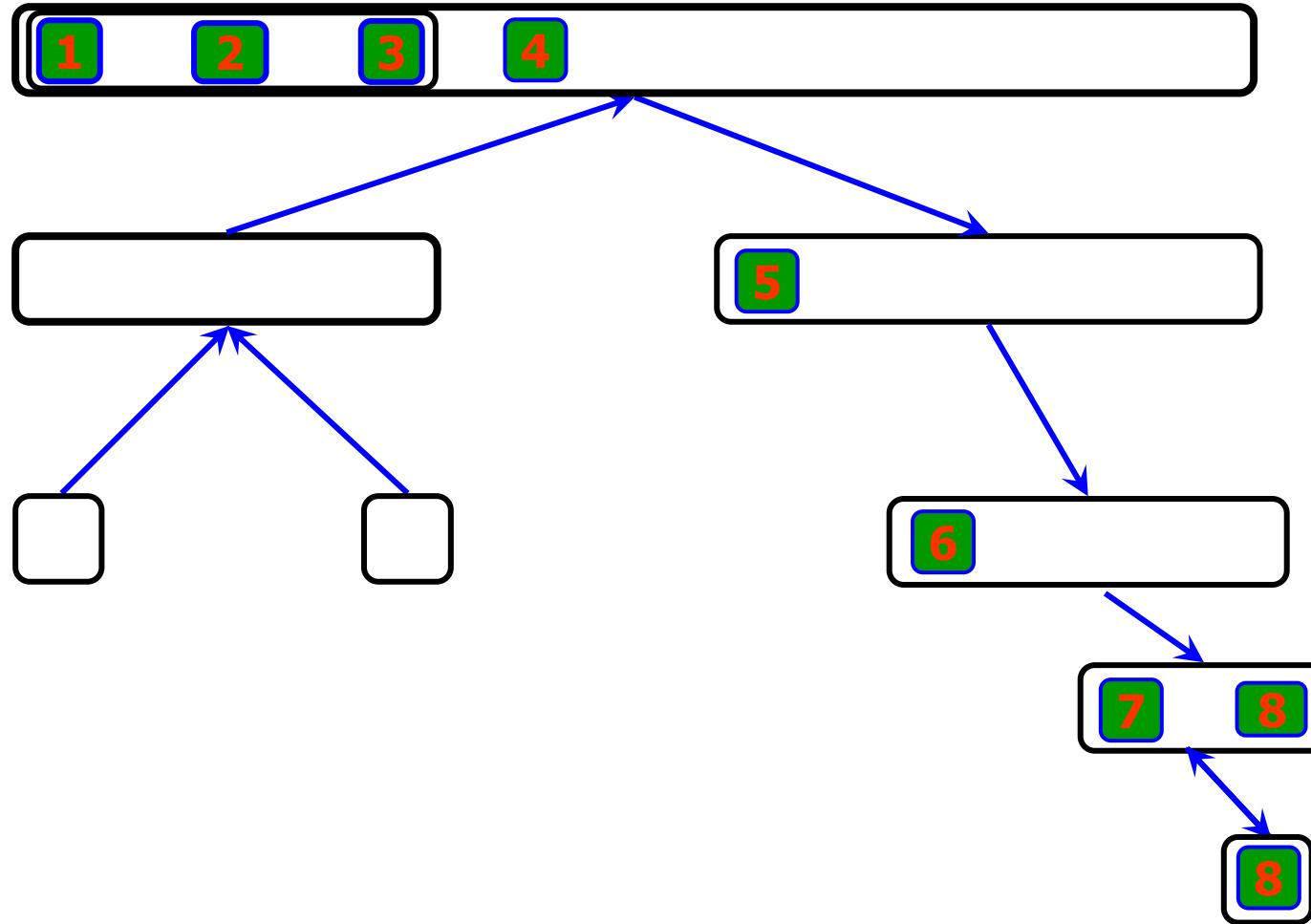
Quick-Sort(A, 0, 7)

Quick-Sort(A, 6, 7), partition



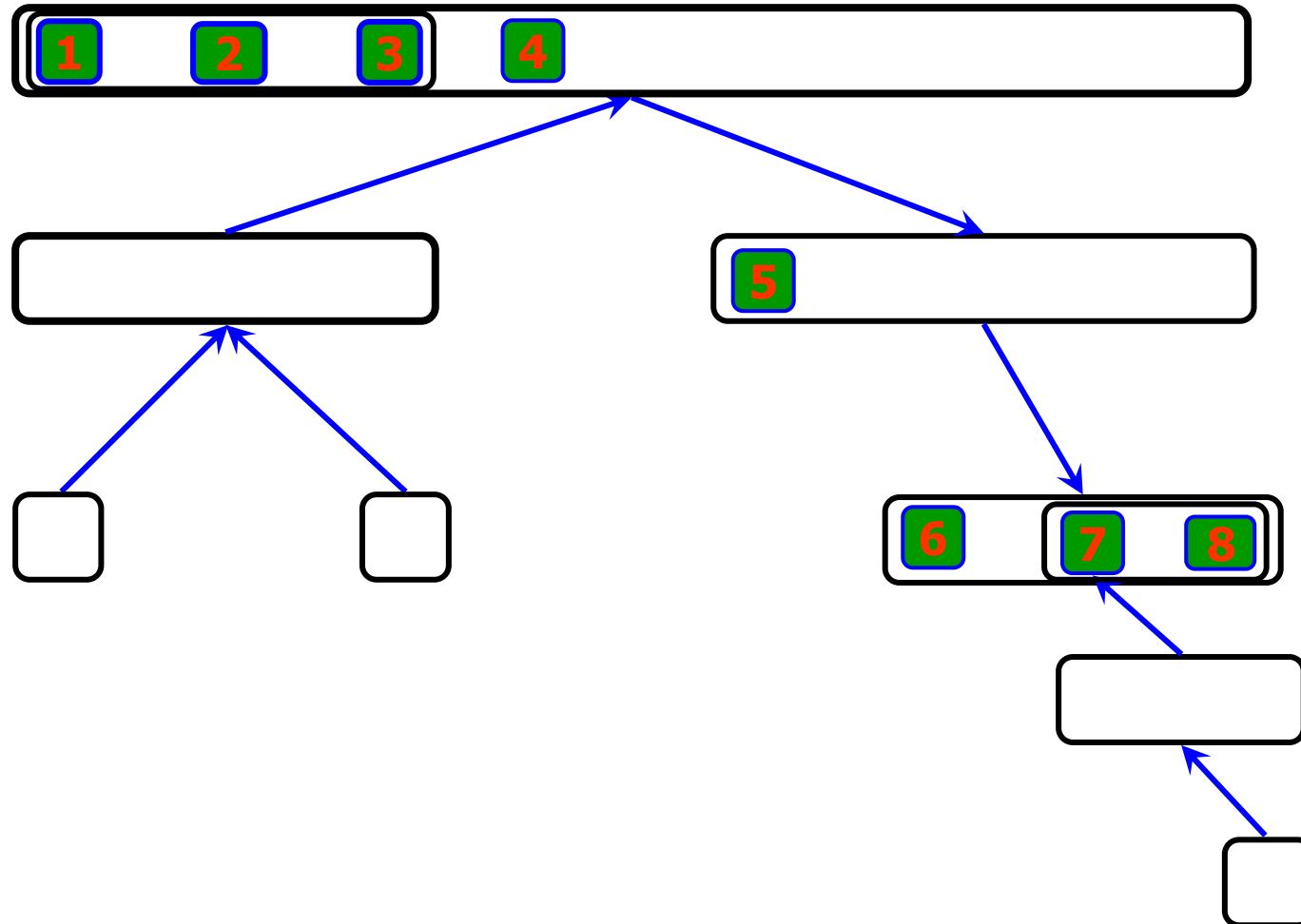
Quick-Sort(A, 0, 7)

Quick-Sort(A, 7, 7) , ~~better case~~



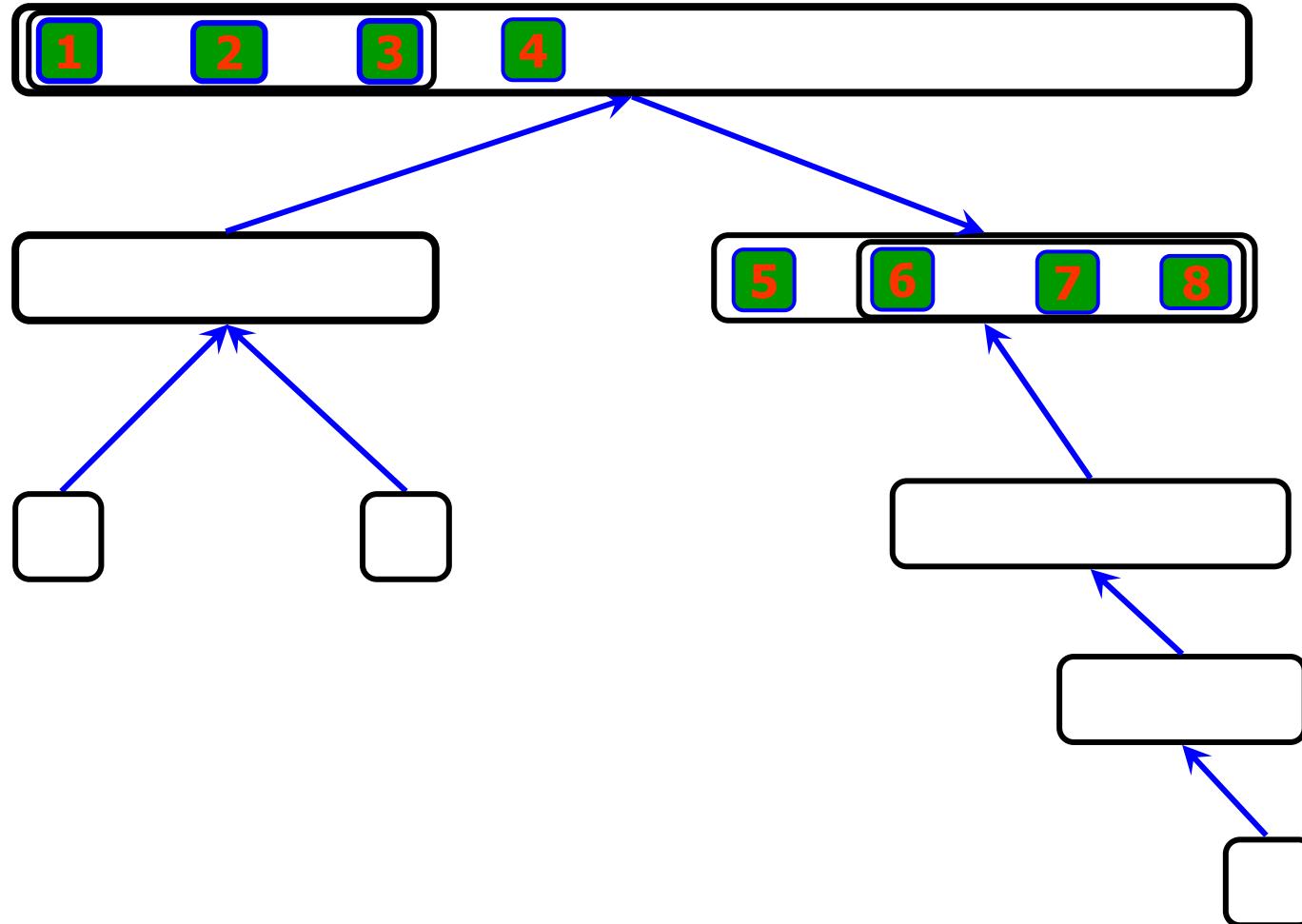
Quick-Sort(A, 0, 7)

Quick-Sort(A, 6, 7) , return



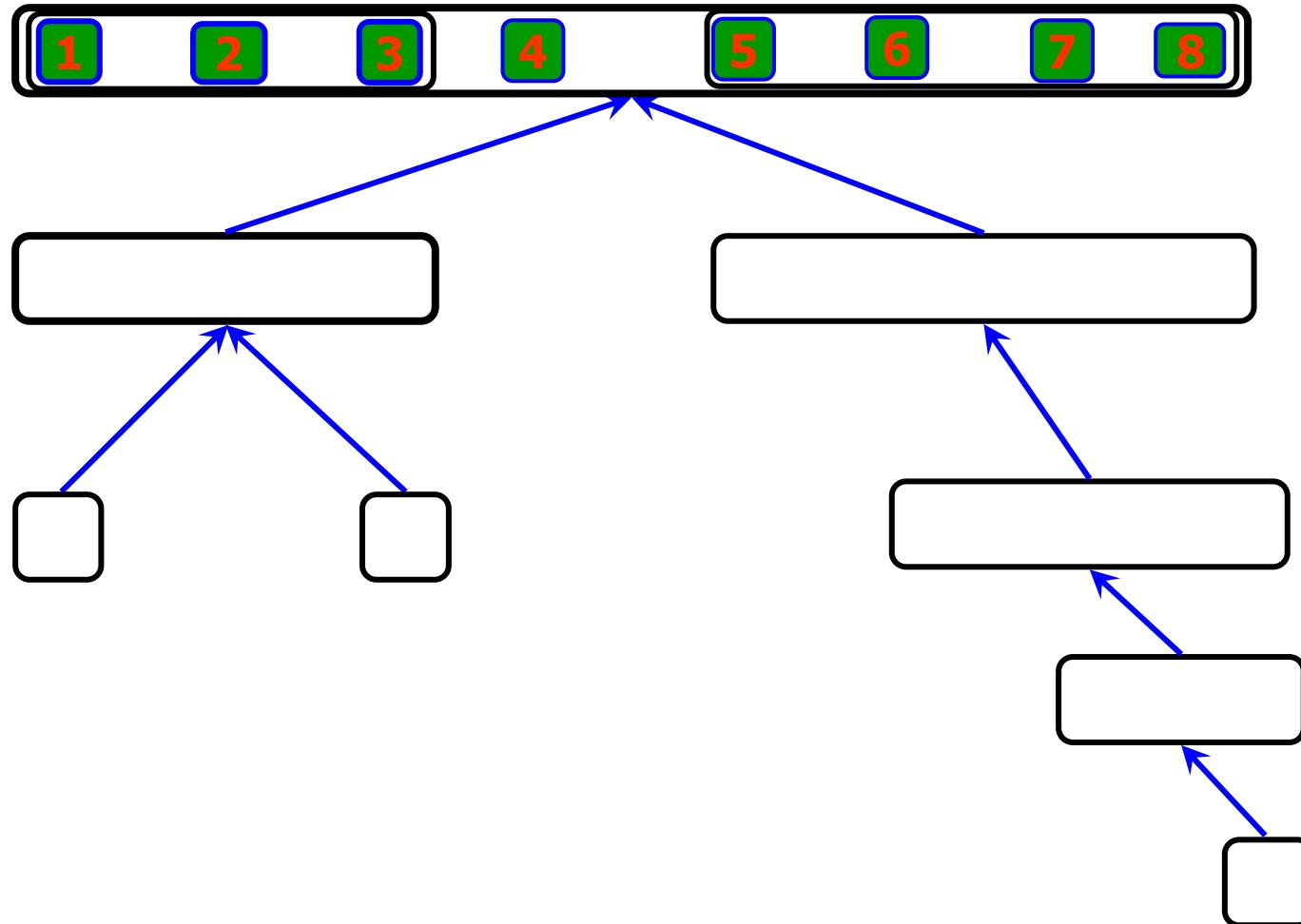
Quick-Sort(A, 0, 7)

Quick-Sort(A, 5, 7) , return



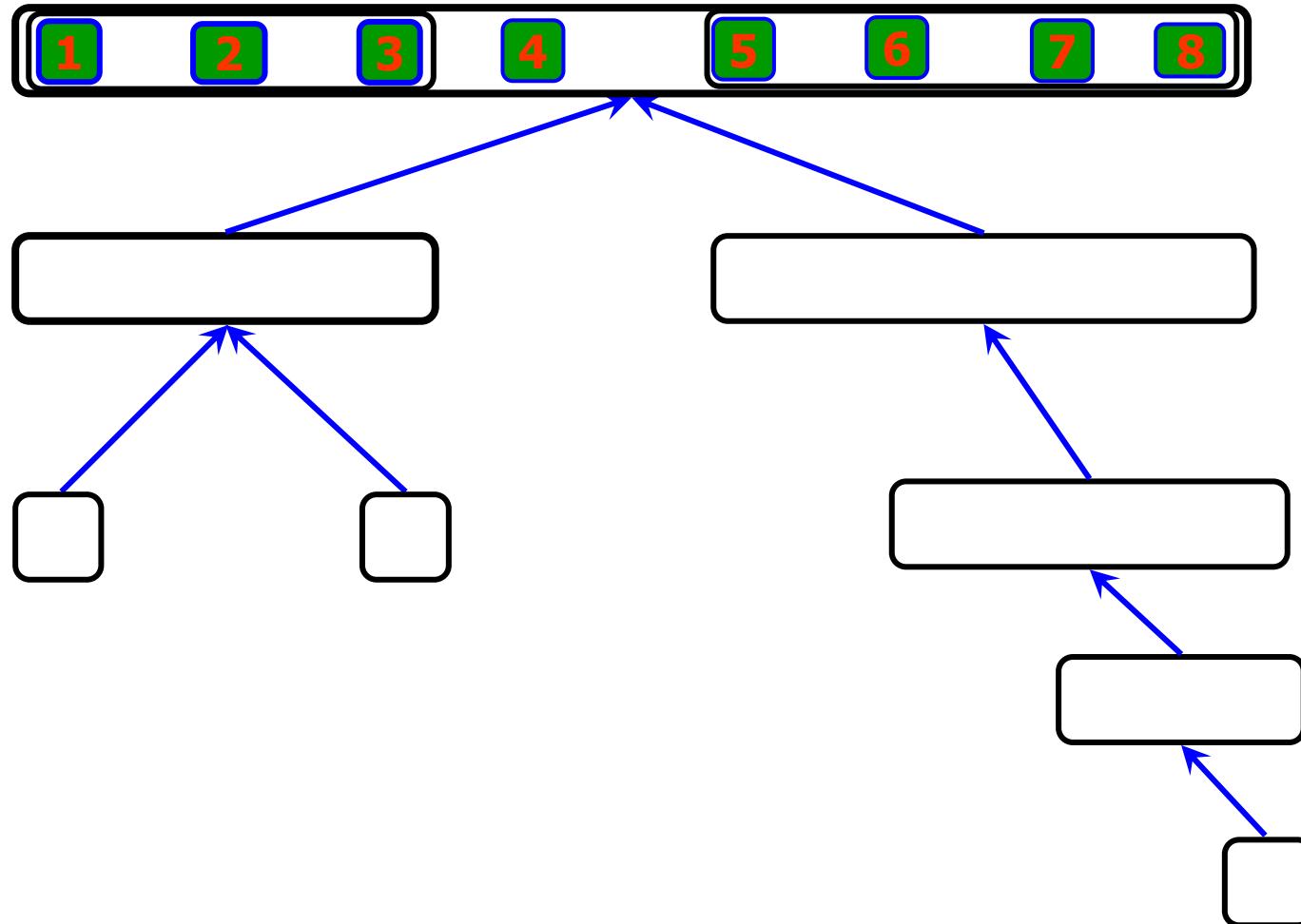
Quick-Sort(A, 0, 7)

Quick-Sort(A, 4, 7) , return



Quick-Sort(A, 0, 7)

Quick-Sort(A, 0, 7) , done!





QUICK SORT ALGORITHM AND CODING

Quick Sort

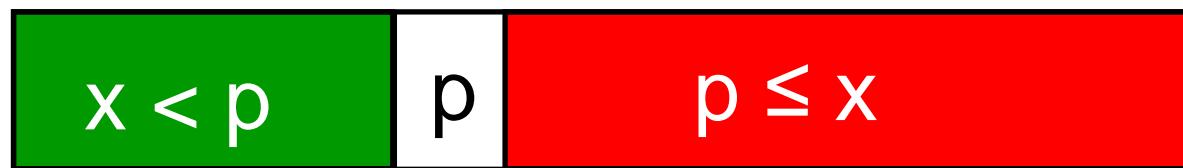
A:
pivot



FirstPart

Partition

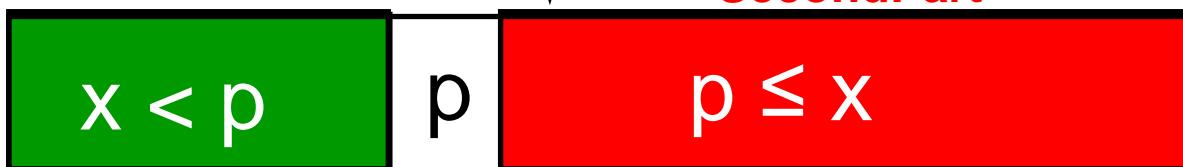
SecondPart



Recursive call

Sorted
FirstPart

Sorted
SecondPart



Sorted

Quick Sort

```
Quick-Sort(A, left, right)
  if      left ≥ right  return
  else
    middle ← Partition(A, left, right)
    Quick-Sort(A, left, middle-1 )
    Quick-Sort(A, middle+1, right)
  end if
```

quickSort

```
# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low --> Starting index,
# high --> Ending index

# Function to do Quick sort
def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr, low, high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

Analysis on Quick Sort

- This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n \log n)$ and $O(n^2)$.
- Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort.
 - QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data.
- However, merge sort is generally considered better when data is huge and stored in external storage.

Summary of key terms

- Dictionary
- divide and conquer
- Merge sort
 - Merge
- Quick sort
 - Partition



LECTURE 6

LINEAR LIST

SEHH2239 Data Structures

Linear Lists

After completing this lesson, you should be able to do the following:

- Use Pointer to represent Linear List
- Implement Linear List Structure and Operations with Python pointer and List

LINEAR LIST ADT

Abstract Data Type (ADT)

- Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.
- The definition of ADT only mentions **what operations** are to be performed but **not how** these operations will be implemented.
- It is called “abstract” because it gives an implementation-independent view.
 - The process of providing only the essentials and hiding the details is known as abstraction.
 - It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

Linear Lists

$$L = (e_0, e_1, e_2, e_3, \dots, e_{n-1})$$

- relationships
- where e_i denotes a list element and list size is n
- $n \geq 0$ is finite
- e_0 is the zero'th (or front) element
- e_{n-1} is the last element
- e_i immediately precedes e_{i+1}

Linear List Examples

- Students in SEHH2239 =
(Jack, Jill, Abe, Henry, Mary, ..., Judy)
- Quizzes in SEHH2239 =
(quiz1, quiz2, quiz3)
- Days of Week = (S, M, T, W, Th, F, Sa)
- Months = (Jan, Feb, Mar, Apr, ..., Nov, Dec)

Linear List Abstract Data Type

AbstractDataType *LinearList*

Operations:

isEmpty(): return `true` iff the list is empty, `false` otherwise

size(): return the size of the list

get(i): return the *i* th element of the list

indexOf(el): return the index of the first occurrence of *el* in
the list, return `-1` if *x* is not in the list

remove(index): remove and return the *index*th element,
elements with higher index have their index reduced by 1

removeNode(Removekey): remove the node with given element
elements with higher index have their index reduced by 1

add(theIndex, x): insert *x* as the *index*th element, elements
with *theIndex >= index* have their index increased by 1

addAtHead(x): inert the *x* at the beginning

addAtTail(x): inert the *x* at the end

listprint(): print the linked list

Linear List Operations—size()

- determine list size

$$L = (a, b, c, d, e)$$

size = 5

Linear List Operations—get(theIndex)

- get element with given index

$$L = (a, b, c, d, e)$$

- $get(0) = a$
- $get(2) = c$
- $get(4) = e$
- $get(-1) = \text{error}$
- $get(9) = \text{error}$

Linear List Operations— indexOf(theElement)

- determine the index of an element

$$L = (a,b,d,b,a)$$

- $\text{indexOf}(d) = 2$
- $\text{indexOf}(a) = 0$
- $\text{indexOf}(z) = -1$

Linear List Operations— remove(theIndex)

- remove and return element with given index

$$L = (a,b,c,d,e,f,g)$$

- *remove(2)* returns *c*
- and *L* becomes *(a,b,d,e,f,g)*

index of *d,e,f,* and *g* decrease by 1

Linear List Operations— remove(theIndex)

- remove and return element with given index

$$L = (a,b,c,d,e,f,g)$$

- *remove(-1)* => error
- *remove(20)* => error

Linear List Operations— remove(theElement)

- remove the first occurrence of the specified element.

$$L = (a,b,c,d,e,f,g)$$

- *remove(c)* and L becomes (a,b,d,e,f,g)
index of $d, e, f,$ and g decrease by 1
- *remove(h)* => no element removed

Linear List Operations— add(theIndex, theElement)

- add an element so that the new element has a specified index

$$L = (a, b, c, d, e, f, g)$$

- $add(0, h) \Rightarrow L = (h, a, b, c, d, e, f, g)$
index of $a, b, c, d, e, f,$ and g increase by 1

Linear List Operations— add(theIndex, theElement)

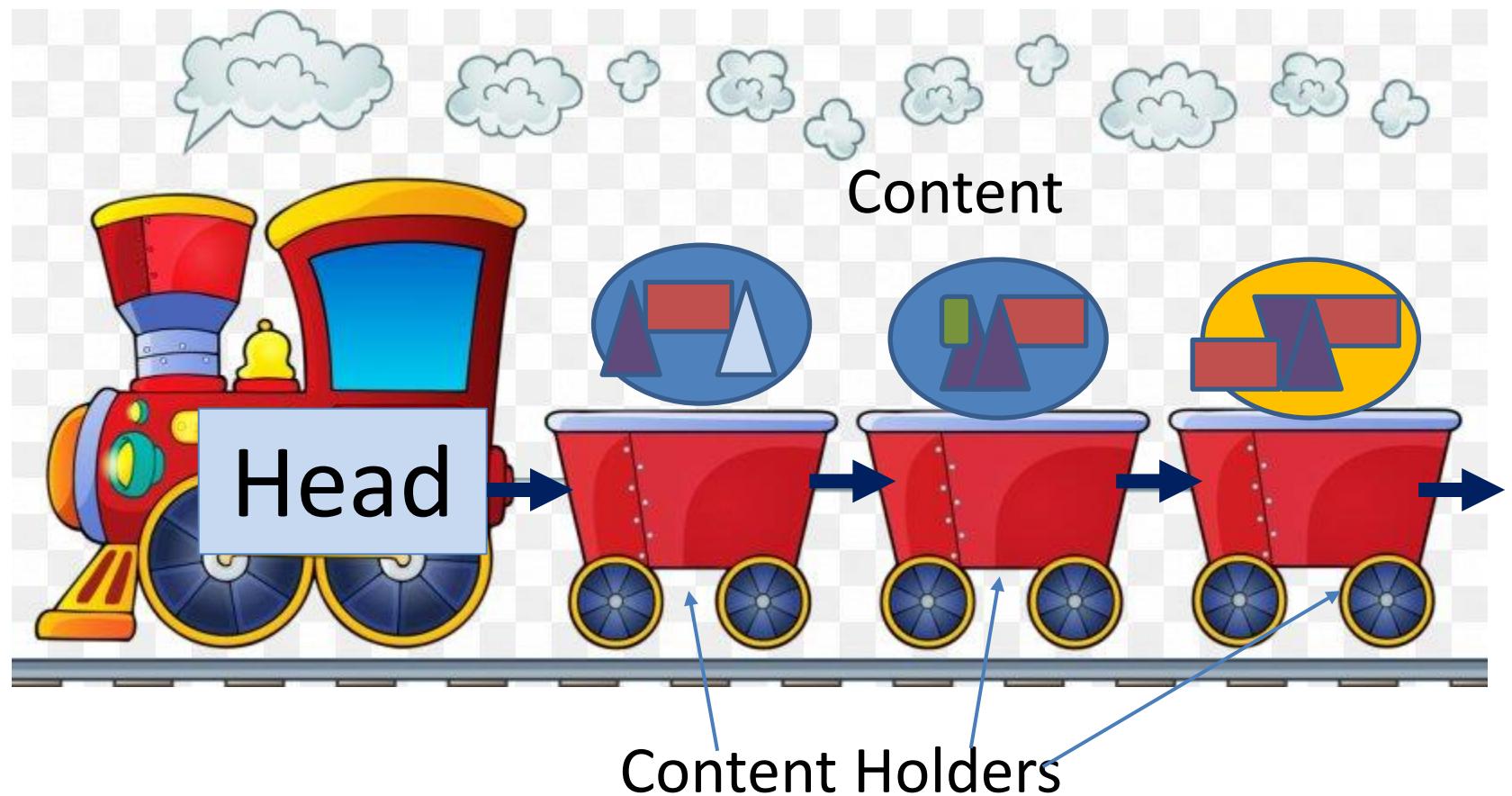
$$L = (a,b,c,d,e,f,g)$$

- $add(2,h) \Rightarrow L = (a,b,h,c,d,e,f,g)$
index of c, d, e, f , and g increase by 1
- $add(10,h) \Rightarrow$ error
- $add(-6,h) \Rightarrow$ error



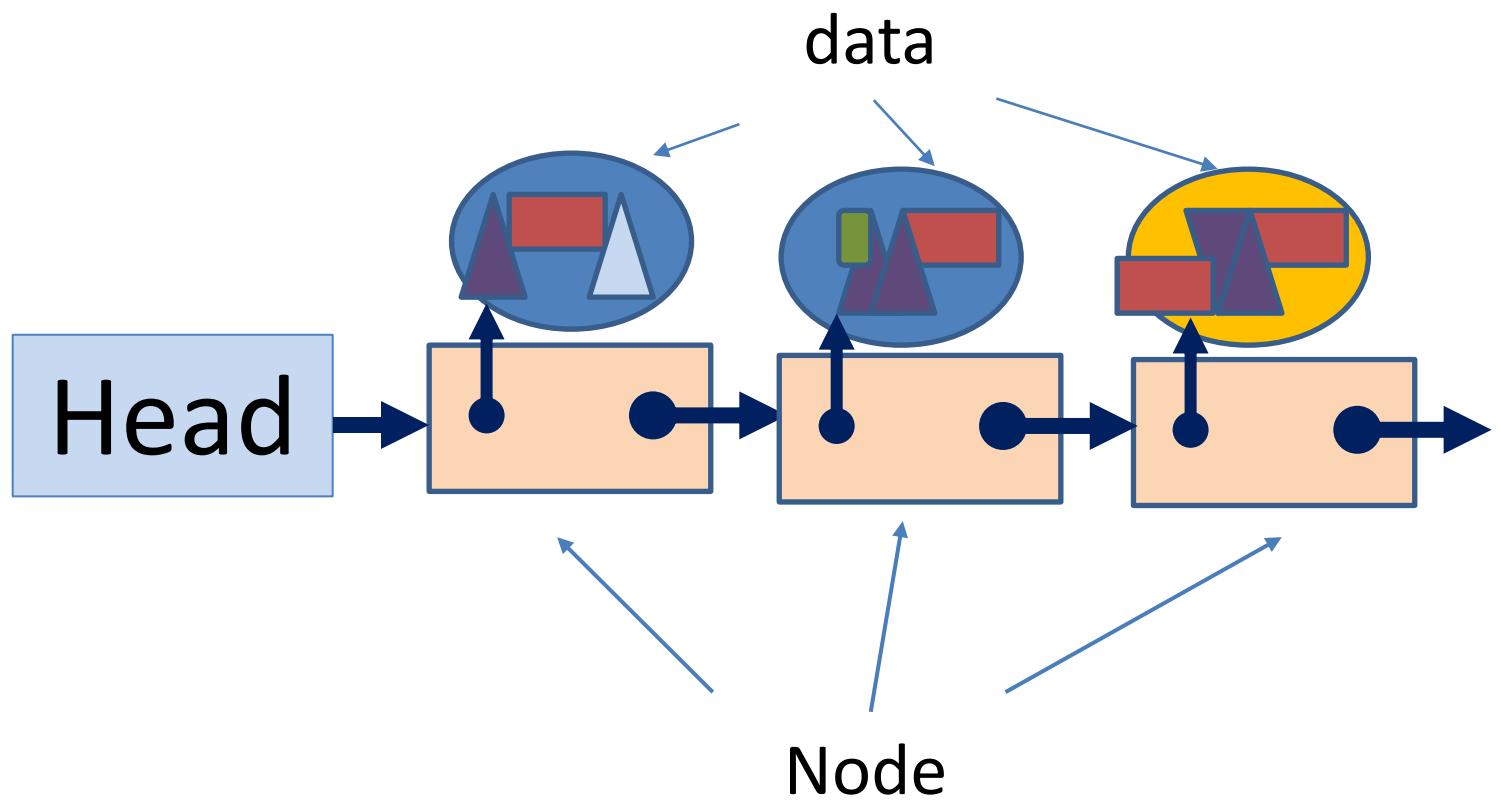
LINEAR LIST IMPLEMENTATION – SINGLY LINKED LIST

Consider a train

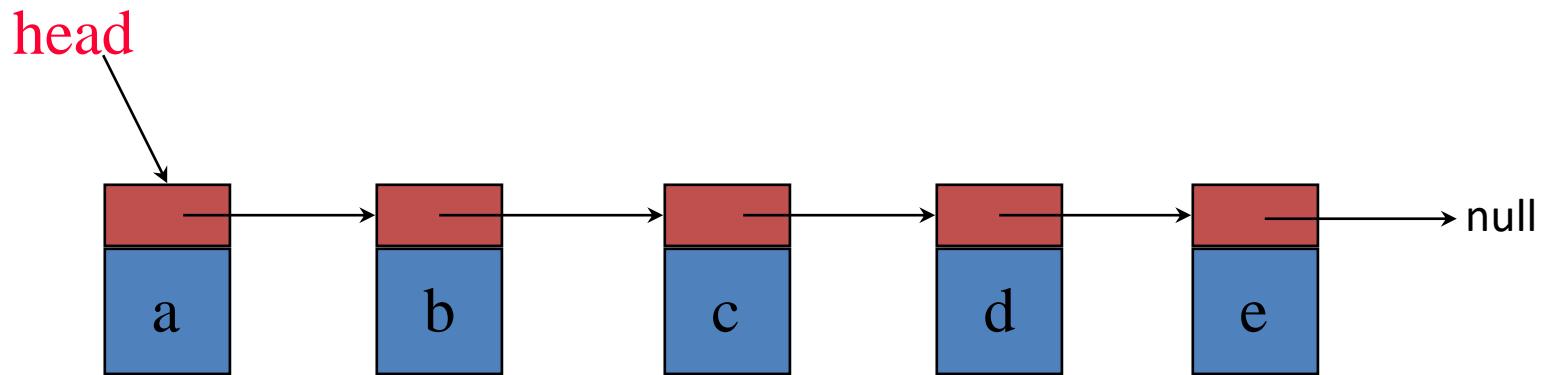


https://favpng.com/png_view/train-train-rail-transport-clip-art-image-png/xu2Tnfr3

A chain of nodes



The Class Node



next



element

size = number of elements

Creation of Linked list



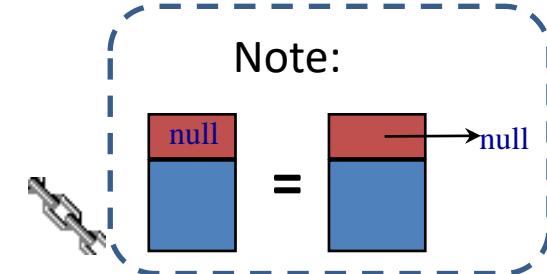
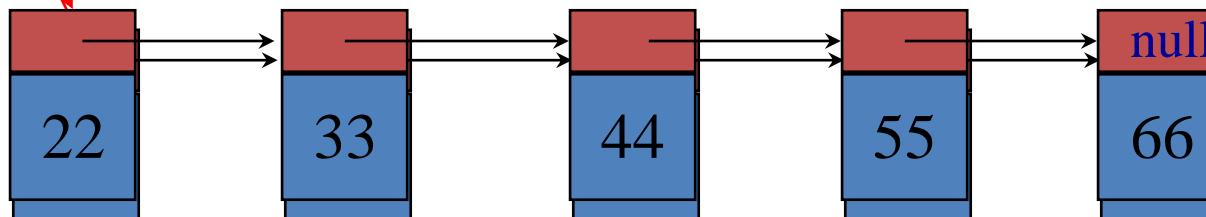
```
class Node:  
    def __init__(self, el = None, n = None):  
        self.next = n  
        self.element = el
```



```
class SLinkedList:  
    def __init__(self):  
        self.head = None
```

LinkedList

head



To implement this chain:

Method 1:

```
list1 = SLinkedList()  
list1.head = Node(22)  
  
list1.head.next = Node(33)  
  
list1.head.next.next = Node(44)  
  
list1.head.next.next.next = Node(55)  
  
list1.head.next.next.next.next = Node(66)
```

```

class Node:
    def __init__(self, el = None, n = None):
        self.next = n
        self.element = el

class SLinkedList:
    def __init__(self):
        self.head = None

    def listprint(self):
        printval = self.head
        while printval is not None:
            print (printval.element)
            printval = printval.next

list1 = SLinkedList()
list1.head = Node(22)

# Link first Node to second node
list1.head.next = Node(33)

# Link second Node to third node.. so on so fore
list1.head.next.next = Node(44)
list1.head.next.next.next = Node(55)
list1.head.next.next.next.next = Node(66)

list1.listprint()

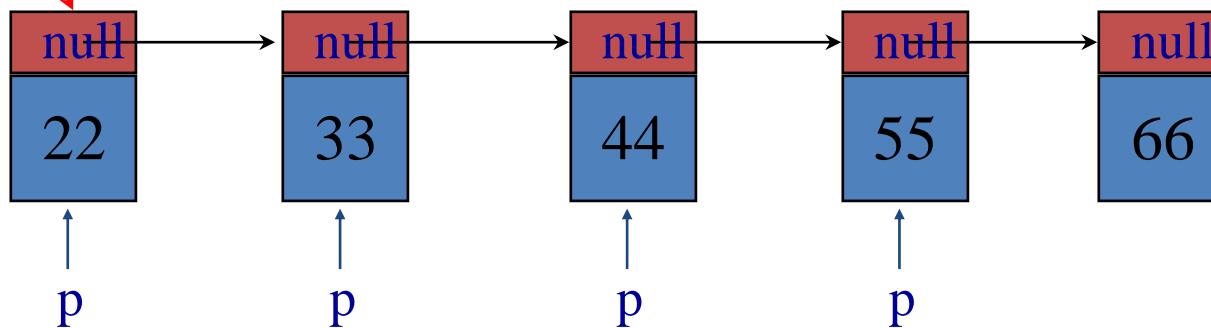
```

22
33
44
55
66

Linked List



head

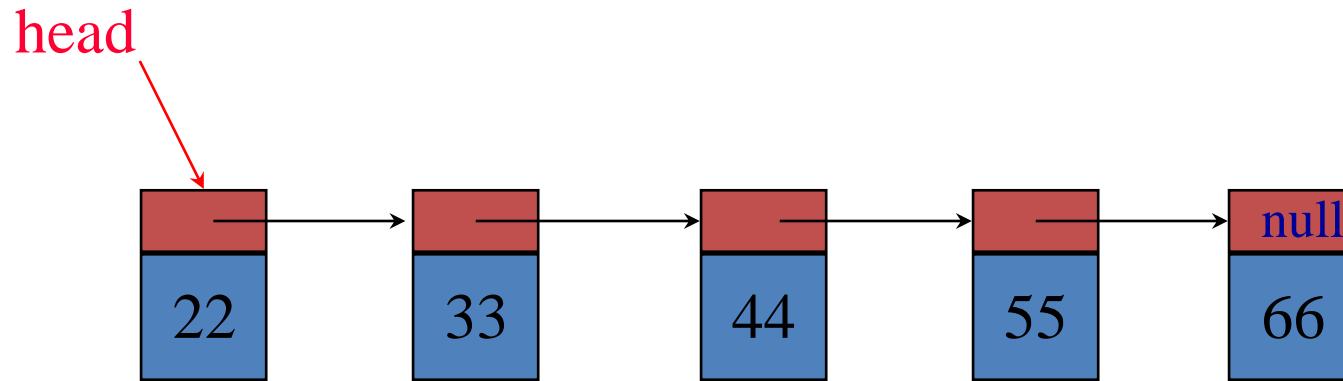


Method 2:

```
list2 = SLinkedList()  
list2.head = Node(22)  
p = list2.head  
p.next = Node(33)  
p = p.next;  
p.next = Node(44)  
p = p.next;  
p.next = Node(55)  
p = p.next;  
p.next = Node(66)
```

SLLEx1.py

Linked List

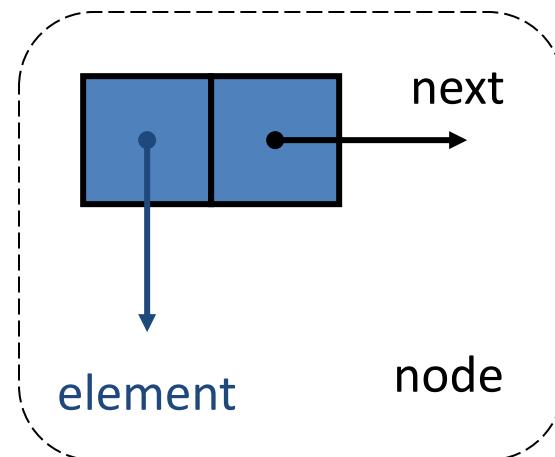
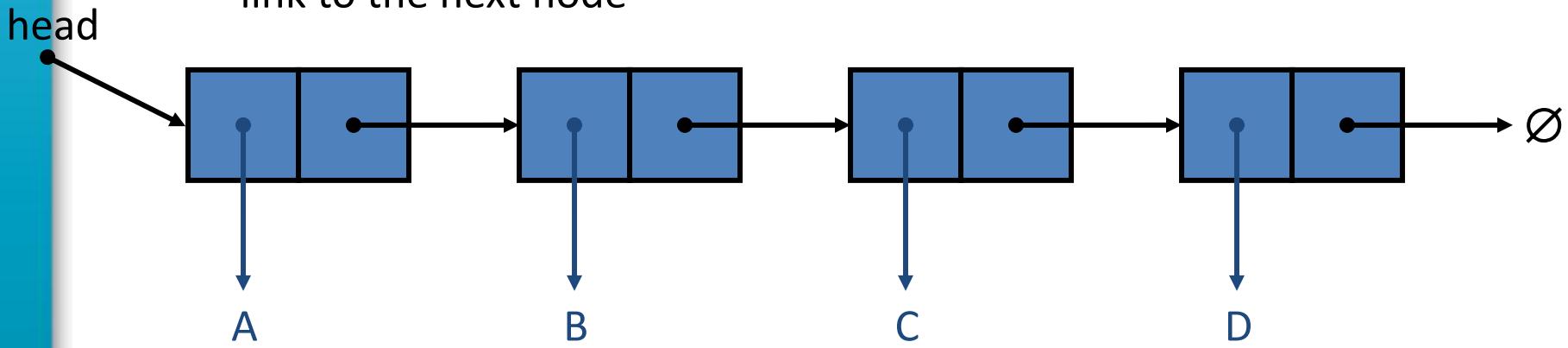


Method 3:

```
array1 = array('i', [33, 44, 55, 66])  
list3 = SLinkedList()  
list3.head = Node(22)  
p = list3.head  
  
for x in array1:  
    p.next = Node(x)  
    p = p.next
```

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
 - element
 - link to the next node



The Class SLinkedList

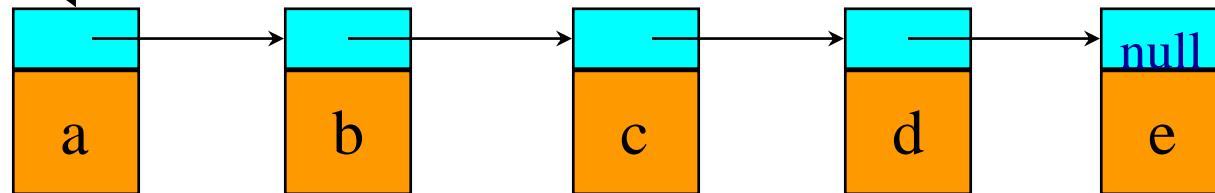
```
class Node:  
    def __init__(self, el = None, n = None):  
        self.next = n  
        self.element = el  
  
class SLinkedList:  
    def __init__(self):  
        self.head = None
```

The Method – isEmpty()

```
#return true iff the list is empty, false otherwise
def isEmpty(self):
    return self.head is None
```

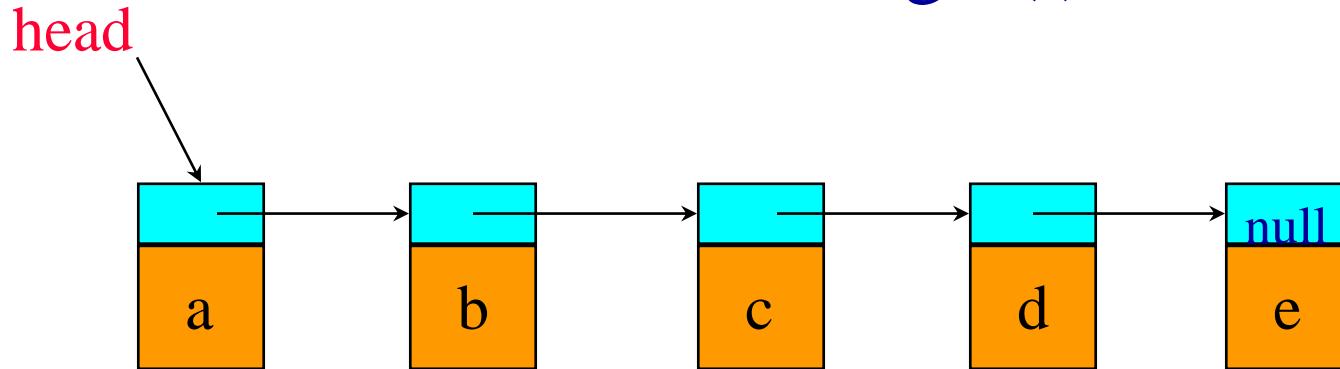
head

The Method - size()



```
#return the size of the list
def size(self):
    size = 0;
    temp = self.head
    while(temp is not None):
        size += 1
        temp = temp.next
    return size
```

The Method – get(i)

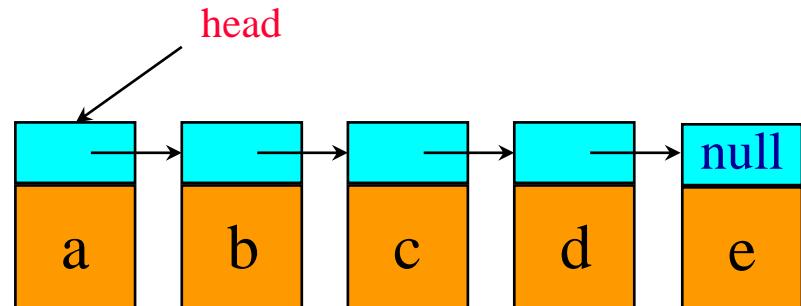


```
#return the i th element of the list
def get(self, i):
    elindex = 0
    temp = self.head
    while temp and elindex != i:
        temp = temp.next
        elindex += 1
    if temp is None:
        return None
    else: return temp.element
```

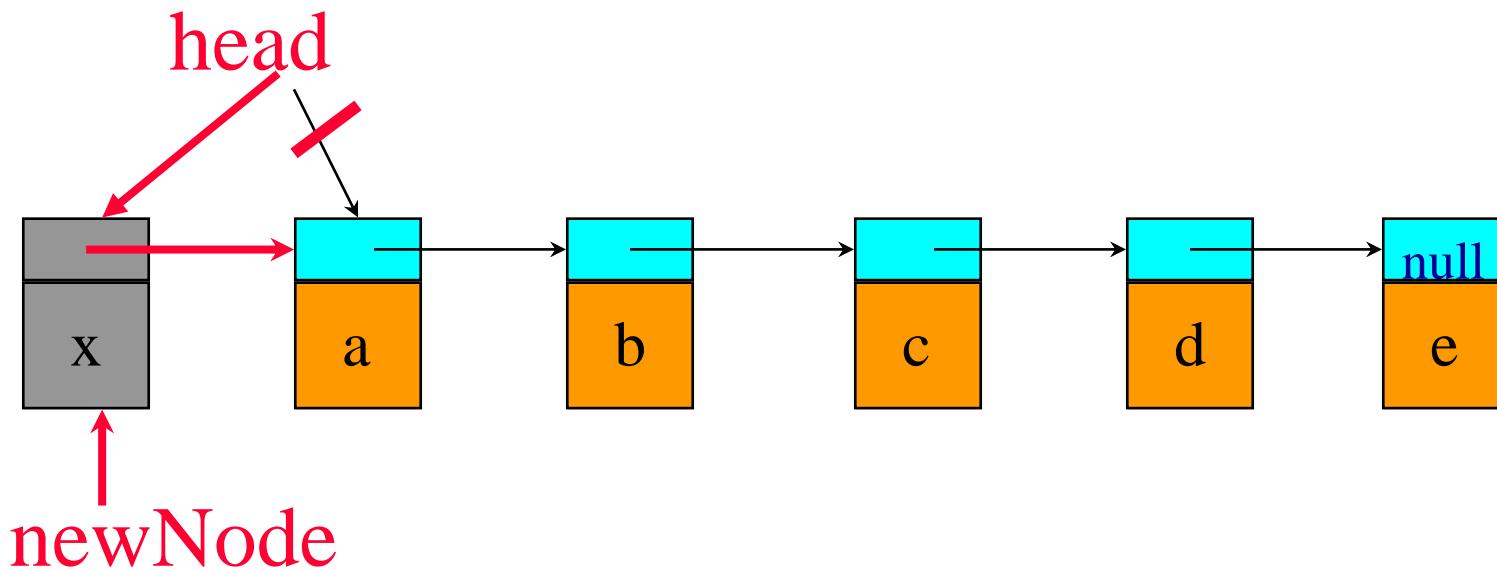
The Method – indexOf(el)

```
#find the index of a node
def indexOf(self, el):
    temp = self.head
    tempindex = 0; #index of temp node
    while temp and temp.element != el:
        #move to the next node
        temp = temp.next
        tempindex += 1

#make sure we found the matching element
if temp is None:
    return -1
else: return tempindex
```

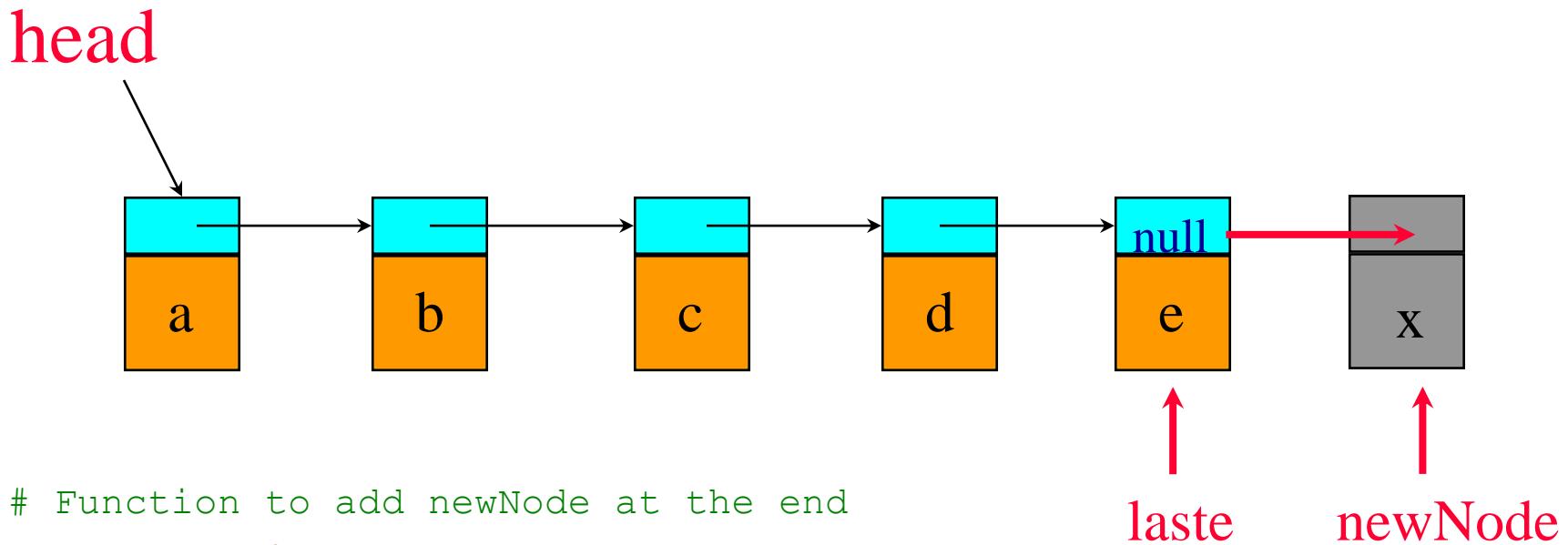


The Method – addAtHead(x)



```
# Inserting at the Beginning
def addAtHead(self, x):
    newNode = Node(x)
    # Update the new nodes next val to existing node
    newNode.next = self.head
    self.head = newNode
```

The Method – addAtTail(x)



```
# Function to add newNode at the end
def addAtTail(self, x):
    NewNode = Node(x)
    if self.head is None: # if only one node
        self.head = NewNode
    return
    #loop to the last node
    laste = self.head
    while(laste.next):
        laste = laste.next
    laste.next=NewNode
```

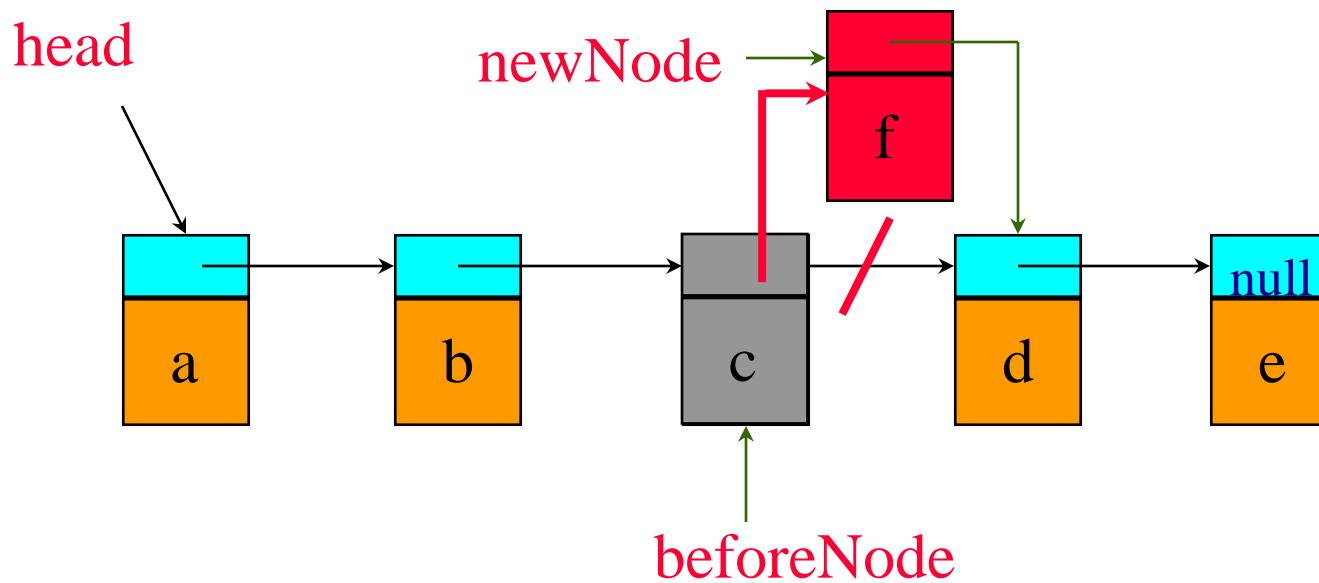
The Method – Add(index, Element)

Add An Element at index 0

```
#insert x as the index th element
def add(self, theIndex, x):
    if theIndex == 0:
        self.addAtHead(x)
```

The Method – Add(index, Element)

Two-Step add(3, 'f')



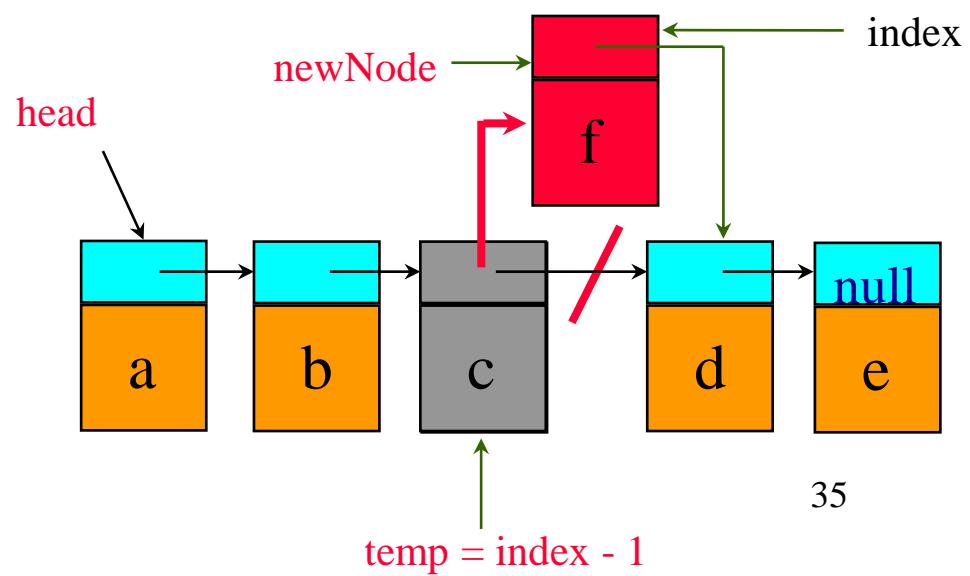
```
beforeNode = head.next.next;
```

```
beforeNode.next = Node('f', beforeNode.next);
```

The Method – Add(index, Element)

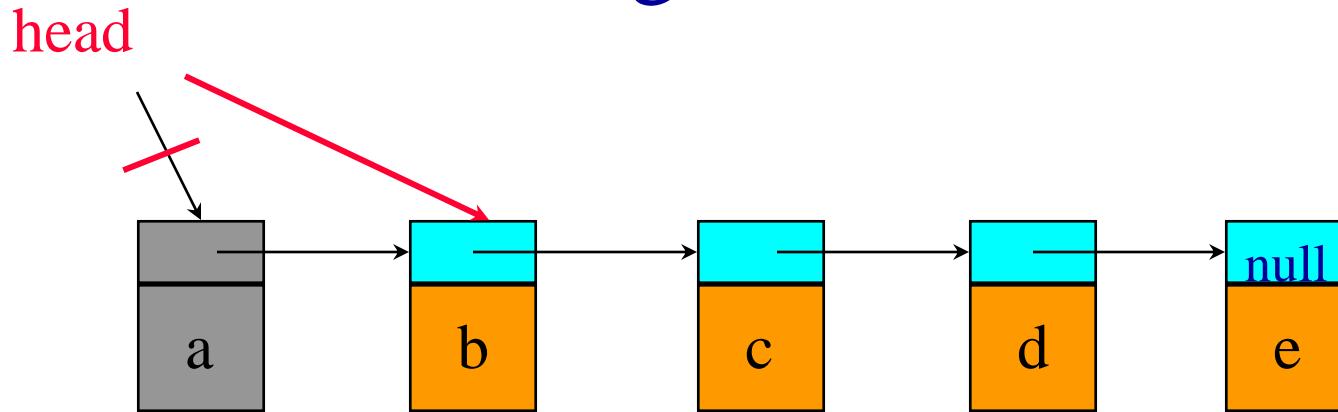
Adding An Element

```
else:  
    #find predecessor of new element  
    temp = self.head  
    elindex = 0  
    while temp and elindex != theIndex-1:  
        temp = temp.next  
        elindex += 1  
    temp.next = Node(x, temp.next)
```



The Method – Remove(index)

Removing the first node

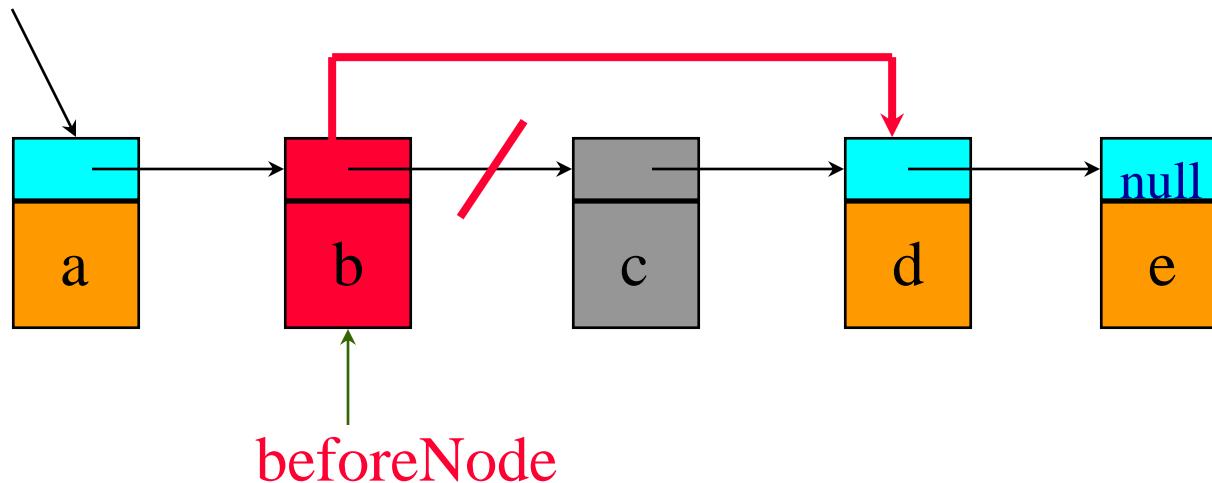


```
def remove(self, index):
    if index == 0:
        if self.isEmpty():
            return None
        ele = self.head.element
        self.head = self.head.next
    return ele
```

The Method – Remove(index)

remove(2)

head



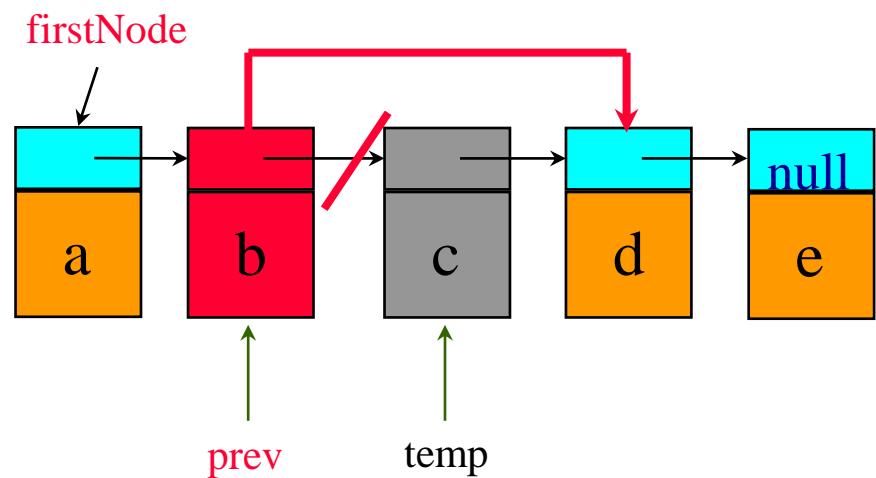
Find **beforeNode** and change its pointer.

beforeNode.next = beforeNode.next.next;

The Method – Remove(index)

remove(2)

```
else:  
    temp = self.head  
    elindex = 0  
    while temp and elindex != index:  
        prev = temp  
        temp = temp.next  
        elindex += 1  
  
    if (temp is None):  
        return None  
  
    prev.next = temp.next  
    ele = temp.element  
    temp = None  
    return ele
```





Remove An Element



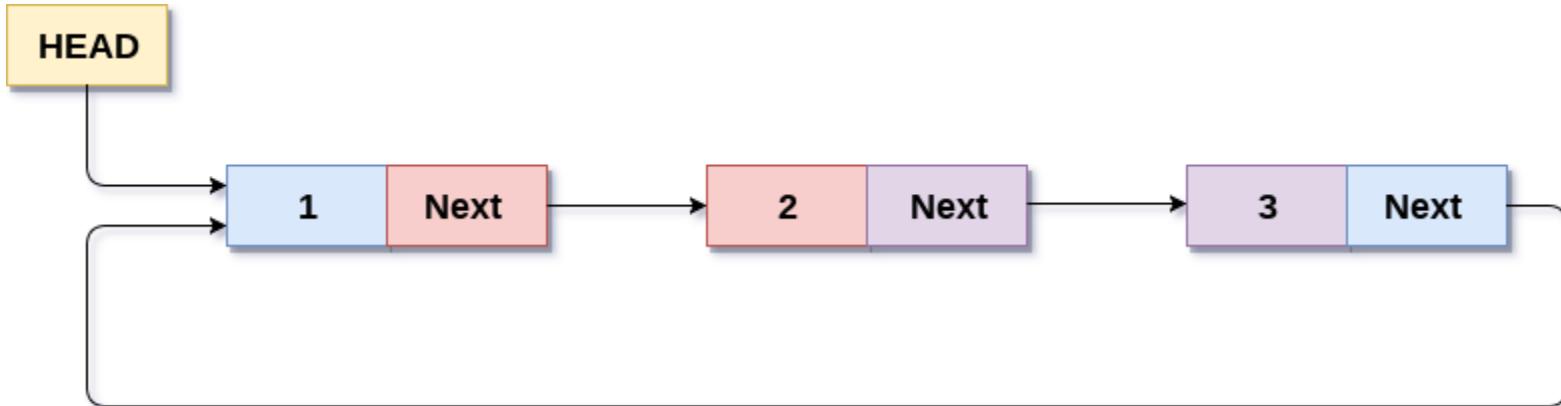
```
# Function to remove node with given element
def removeNode(self, Removekey):
    temp = self.head

    if (temp is not None):
        if (temp.element == Removekey): #remove the first node
            self.head = temp.next
            temp = None
            return
        while (temp is not None):
            if temp.element == Removekey: #found the node
                break
            prev = temp
            temp = temp.next

        if (temp is None): #no match node is found
            return

    prev.next = temp.next #remove the node
    temp = None
```

Circular linked lists



Circular Singly Linked List

Advantage of circular linked list

- Entire list can be traversed from any node of the list.
- It saves time when we have to go to the first node from the last node.
- Its is used for the implementation of **queue**.
- Reference to previous node can easily be found.
- When we want a list to be accessed in a circle or loop then circular linked list are used.



REVIEW ON LIST

List in Python

- The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets.
- Important thing about a list is that items in a list need not be of the same type.

List

```
list1 = ['HKCC', 'CPCE', 19, 2000]
list2 = [1, 2, 3, 4, 5]
list3 = ["a", "b", "c", "d"]

print("list1[0]: ", list1[0])
print("list2[1:5]: ", list2[1:5])
```

```
list1[0]: physics
list2[1:5]: [2, 3, 4, 5]
```

More operations in List

```
print("Size: ", len(list1))
print(3 in list2)
for x in list3:
    print(x)

del list1[2]
print("After deleting value at index 2 : ")
print(list1)
```

```
Size: 4
True
a
b
c
d
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

More operations in List

```
print("After appending new item orange in the list: ")  
list1.append("orange")  
print(list1)
```

```
print("After Adding value HKCC at index 1 : ")  
list1.insert(1, "HKCC")  
print(list1)
```

```
print("Add the elements of list3 to list2:: ")  
list2.extend(list3)  
print(list2)
```

```
After appending new item orange in the list:  
['physics', 'chemistry', 2000, 'orange']  
After Adding value HKCC at index 1 :  
['physics', 'HKCC', 'chemistry', 2000, 'orange']  
Add the elements of list3 to list2::  
[1, 2, 3, 4, 5, 'a', 'b', 'c', 'd']
```

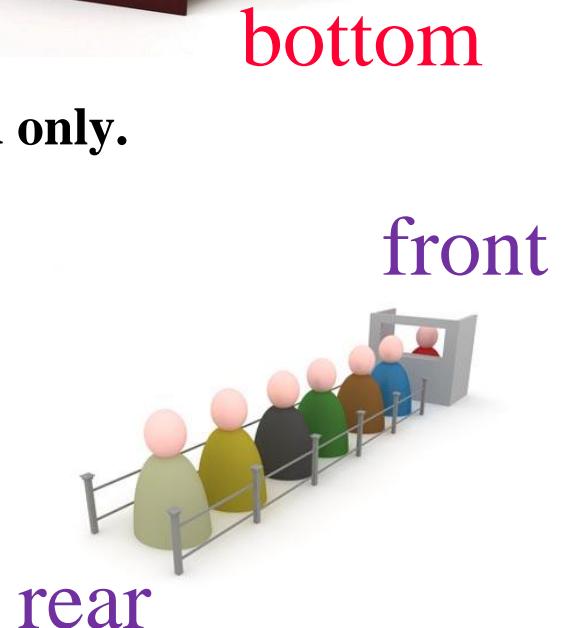
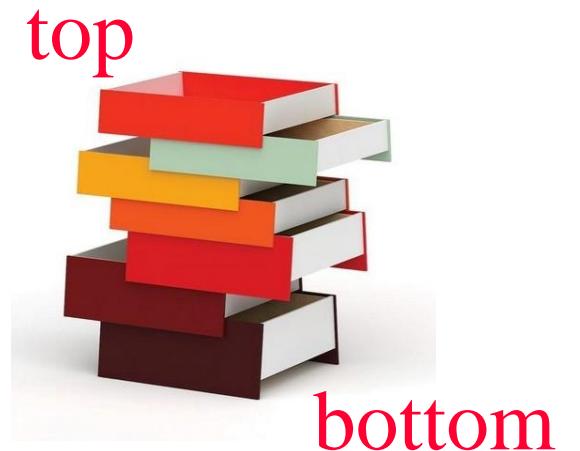
SEHH2239

Data Structures

Lecture 7

Stacks and Queue

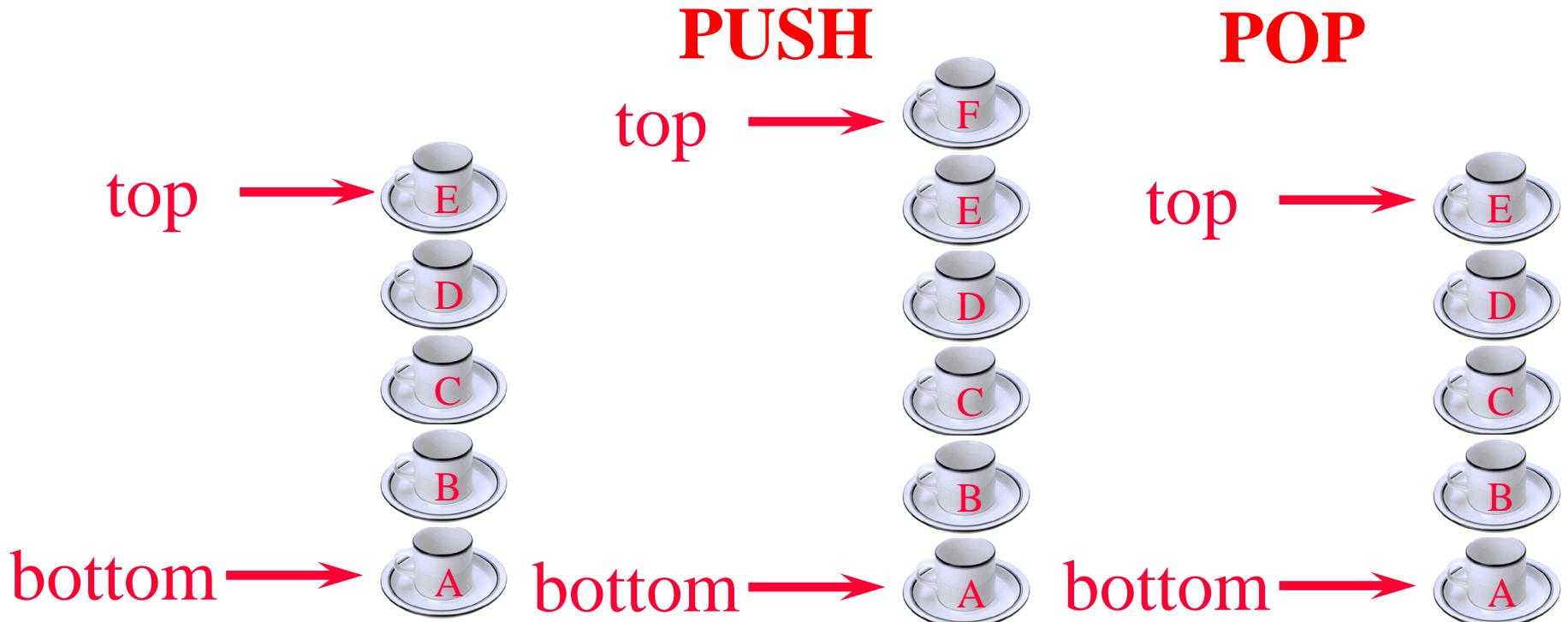
- Stacks
 - Linear list.
 - One end is called **top**.
 - Other end is called **bottom**.
 - **Additions to and removals from the top end only.**
- Queue
 - Linear list.
 - One end is called **front**.
 - Other end is called **rear**.
 - **Additions are done at the rear only.**
 - **Removals are made from the front only.**



Stack



Stack Of Cups



- Add a cup to the stack. (F is added) -- **PUSH**
- Remove a cup from the stack. (F is removed) -- **POP**
- A stack is a **LIFO** list. (Last In First Out)

The Stack Abstract Data Type

Idea: If we enforce the
LIFO principle, it becomes a stack.

A stack S is an **abstract data type (ADT)** that supports following two fundamental methods:

`push(o)`: Insert object o at the top of the stack

Input: Object; **Output**: None.

`pop()`: Remove from the stack and return the top object on
the stack; an error occurs if the stack is empty.

Input: None; **Output**: Object

The Stack Abstract Data Type

Other supporting methods:

peek() / top(): Return the top object on the stack, without removing it; an error occurs if the stack is empty.

Input: None; Output: Object

empty() / isEmpty(): Return a Boolean indicating if the stack is empty.

Input: None; Output: Boolean

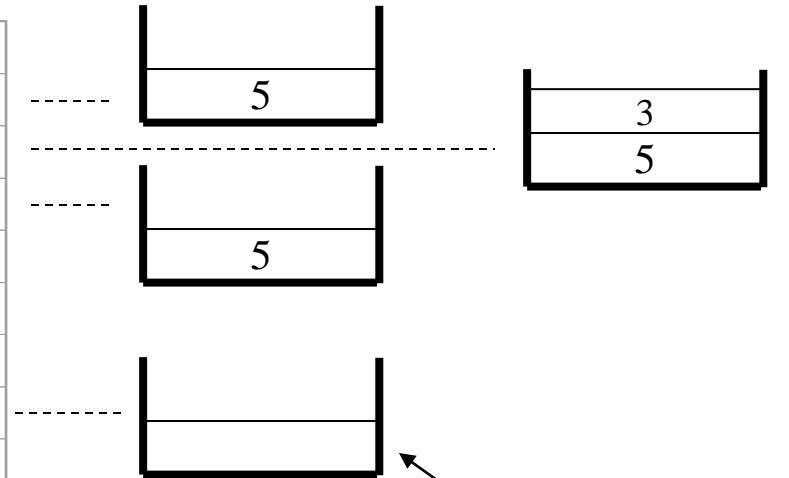
Optional

size(): Return the number of objects in the stack.

Input: None; Output: Integer

This table shows a series of stack operations and their effects.
The stack is initially empty.

Operation	Output	S
push(5)	-	(5)
push(3)	-	(5,3)
pop()	3	(5)
push(7)	-	(5,7)
pop()	7	(5)
top()	5	(5)
pop()	5	()
pop()	“error”	()
isEmpty()	true	()
push(9)	-	(9)
push(7)	-	(9,7)
push(3)	-	(9,7,3)
push(5)	-	(9,7,3,5)
size()	4	(9,7,3,5)
pop()	5	(9,7,3)
push(8)	-	(9,7,3,8)
pop()	8	(9,7,3)
pop()	3	(9,7)



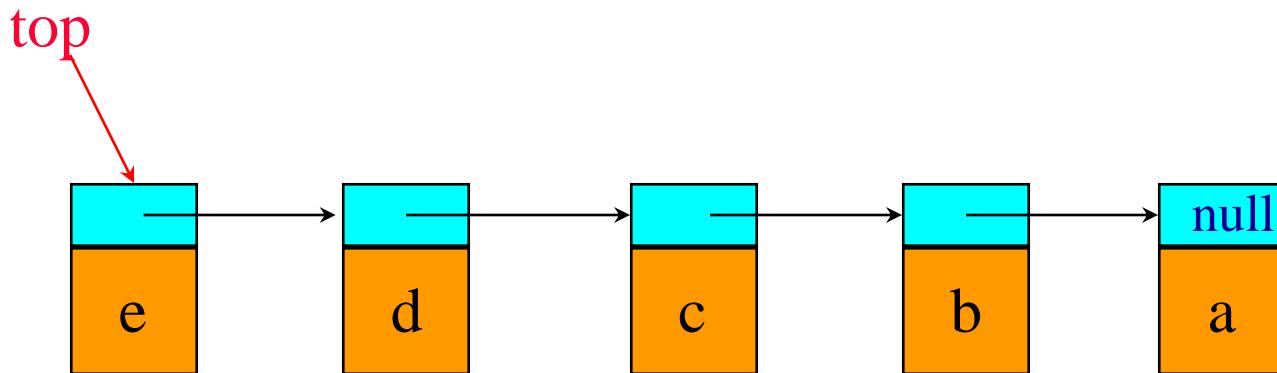
The stack is
now empty.

Linked Chain Implementation of Stack

Linked Implementation

- Use the **Node** class to create an element in stack.
- Use a pointer **top** to point at the top element.
 - Stack elements are in **Node** objects.
 - Top element is in **top.element**.
 - Bottom element is in **top.next...** structure
 - e.g. if there are five elements in stack, **size** is 5, the bottom element is represented by **top.next.next.next**.

Linked Implementation

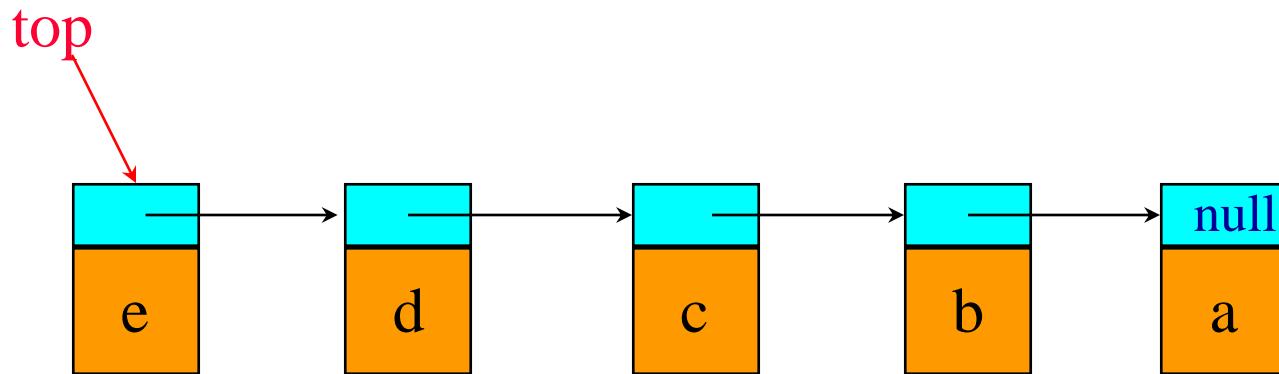


```
class Node:  
    def __init__(self, el = None, n = None):  
        self.next = n  
        self.element = el
```

```
class LinkedStack:  
    def __init__(self):  
        self.top = None  
        self.size = 0
```

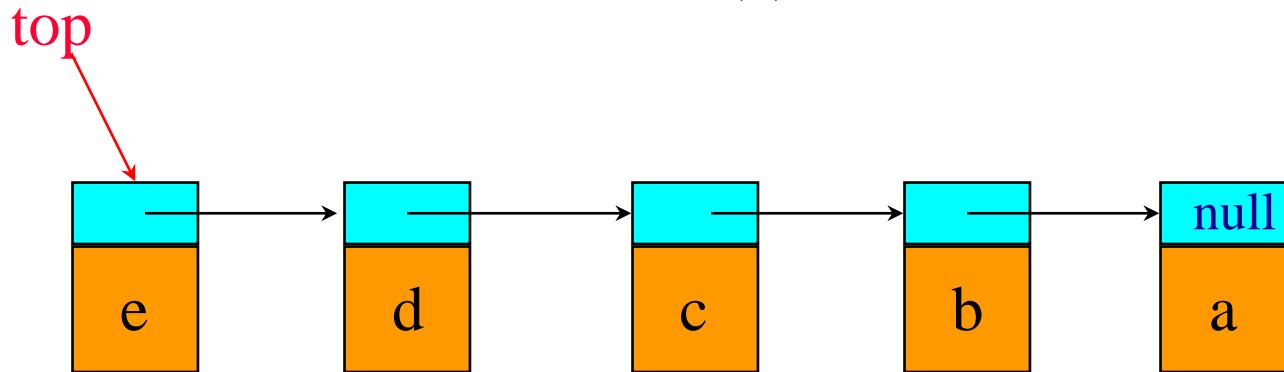
LinkedStack.py

push(...)



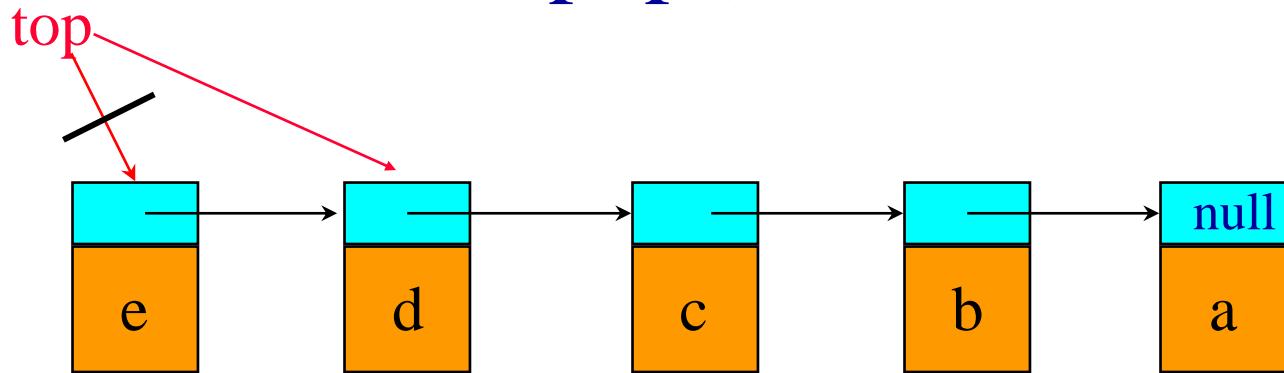
```
#add theElement to the top of the stack
def push(self, element):
    newNode = Node(element)
    newNode.next = self.top
    self.top = newNode
    self.size += 1
```

peek()



```
#return top element of stack
def peek(self):
    if (self.empty()):
        return ("Empty Stack")
    else:
        return self.top.element;
```

pop()



```
#remove top element of stack and return it
def pop(self):
    if (self.empty()):
        return ("Empty Stack")
    topelement = self.top.element
    self.top = self.top.next
    self.size -= 1
    return topelement
```

empty()

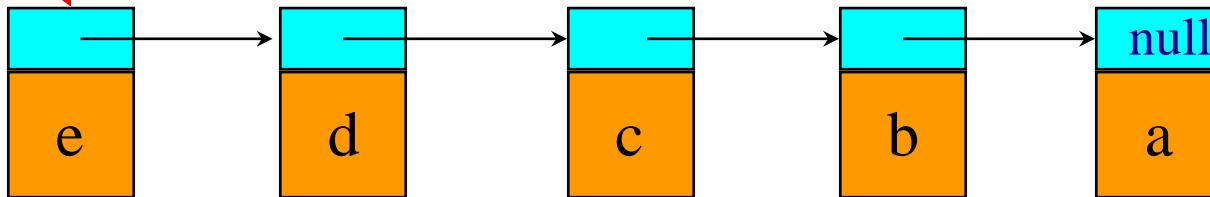
top
↓
null

size = 0

```
#return true if the stack is empty
def empty(self):
    return self.size == 0
```

stack_size()

top



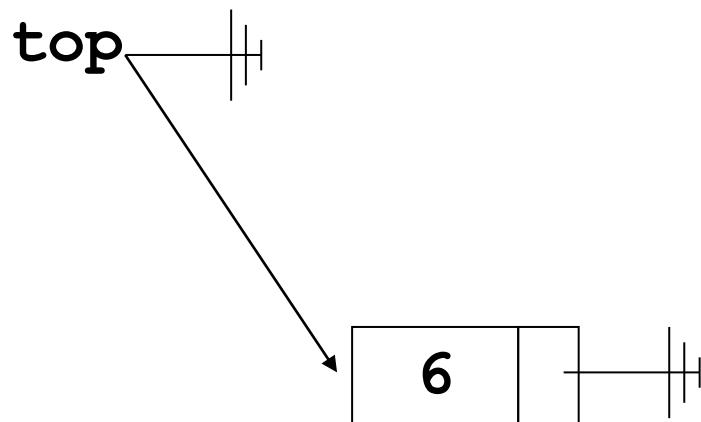
size = 5

```
#return the size of the stack
def stack_size(self):
    return self.size
```

Linked Stack Example

Python Code

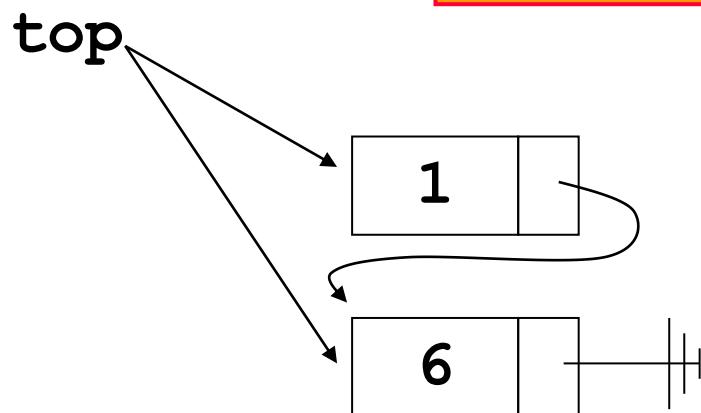
```
st = LinkedStack()  
st.push(6);
```



Linked Stack Example

Python Code

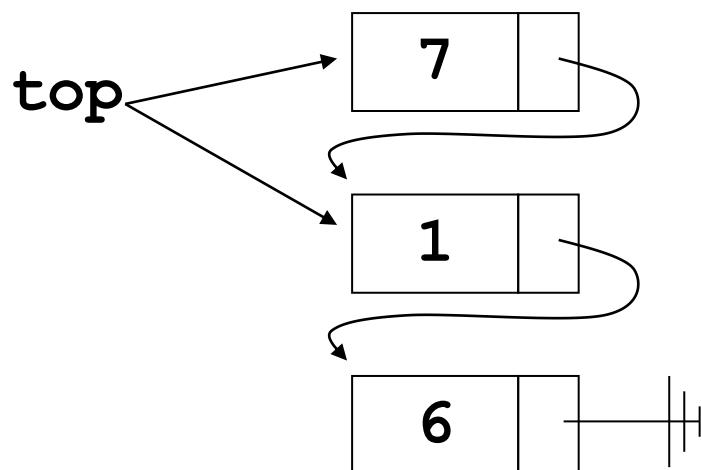
```
st = LinkedStack()  
st.push(6);  
st.push(1);
```



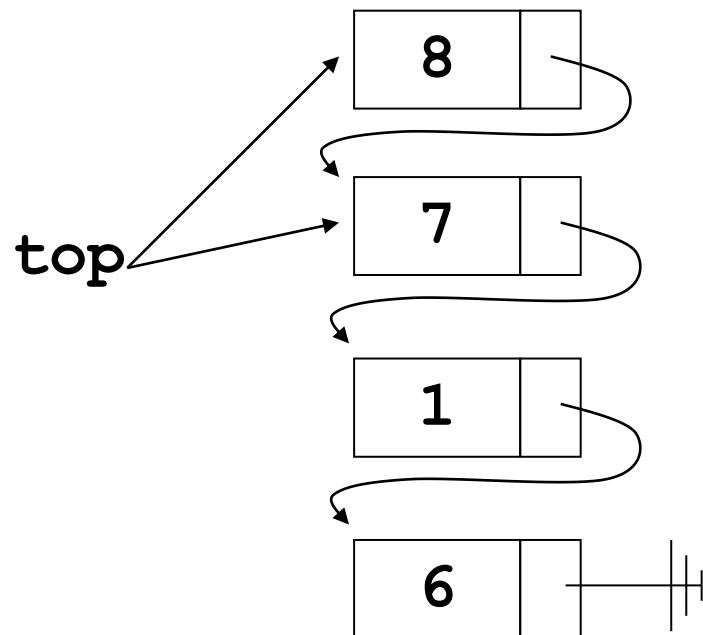
Linked Stack Example

Python Code

```
st = LinkedStack()  
st.push(6);  
st.push(1);  
st.push(7);
```



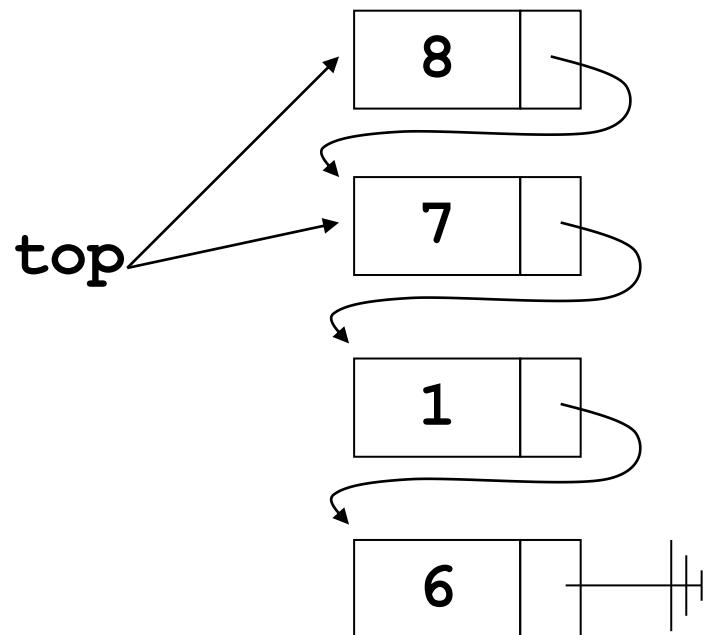
Linked Stack Example



Python Code

```
st = LinkedStack()  
st.push(6);  
st.push(1);  
st.push(7);  
st.push(8);
```

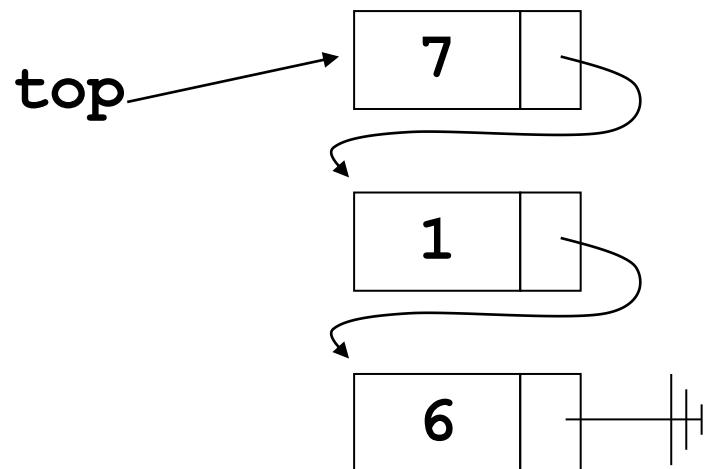
Linked Stack Example



Python Code

```
st = LinkedStack()  
st.push(6);  
st.push(1);  
st.push(7);  
st.push(8);  
st.pop();
```

Linked Stack Example



Python Code

```
st = LinkedStack()  
st.push(6);  
st.push(1);  
st.push(7);  
st.push(8);  
st.pop();
```

Array Implementation of Stack

Array Implementation

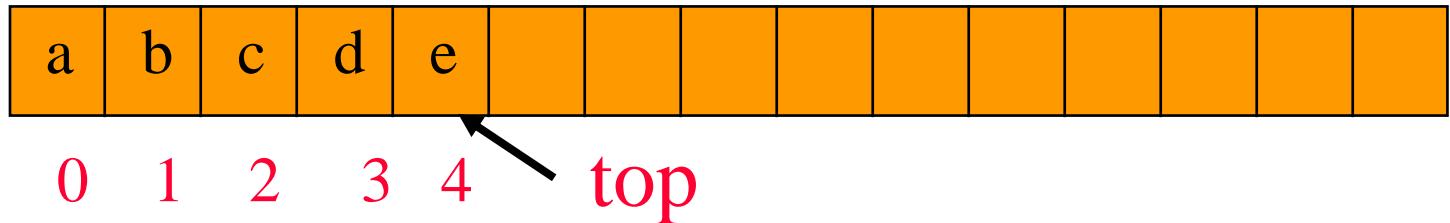
- Use a one-dimensional array `stack` whose data type is `Object`.
- Use an `int` variable `size` to indicate the number of elements in stack.
 - Stack elements are in `stack[0:size-1]`.
 - Top element is in `stack[size-1]`.
 - Bottom element is in `stack[0]`.
 - Stack is empty iff `size = 0`.

Array Implementation

```
class ArrayStack:  
    def __init__(self):  
        self.stack = []  
        self.size = 0
```

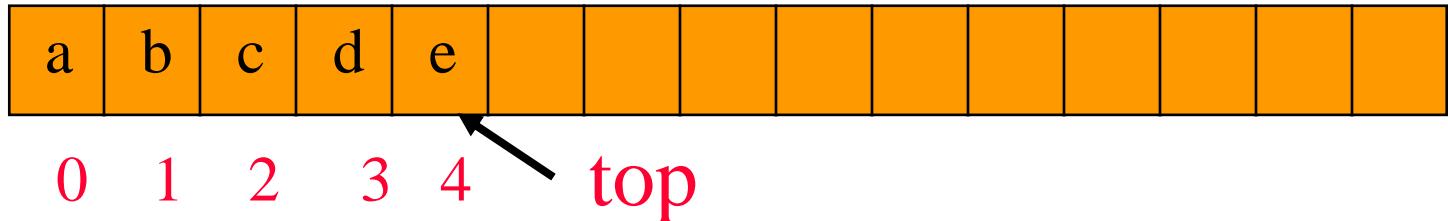
ArrayType.py

push(...)



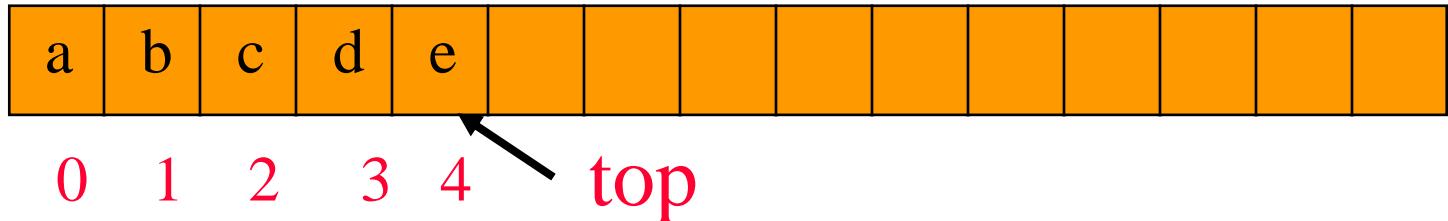
```
#add theElement to the top of the stack
def push(self, element):
    self.stack.append(element)
    self.size += 1
```

peek()



```
# Use peek to look at the top of the stack
def peek(self):
    if(self.size > 0):
        #The last element in the stack
        return self.stack[1]
    else:
        return ("Empty Stack")
```

pop()



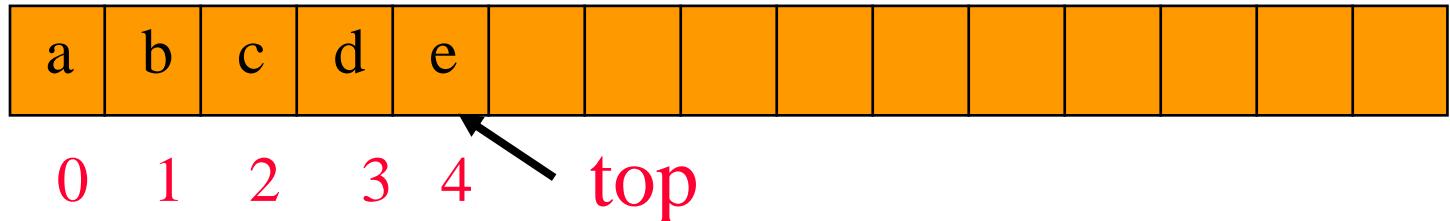
```
# Use list pop method to remove element
def pop(self):
    if (self.size > 0):
        self.size -= 1
        return self.stack.pop()
    else:
        return ("Empty Stack")
```

empty()



```
#return true if the stack is empty
def empty(self):
    return self.size == 0
```

size()



size = 5

```
#return the size of the stack
def stack_size(self):
    return self.size
```

Applications of Stacks

Applications of Stacks

- Call stack (recursion).
- Searching networks, traversing trees (keeping a track where we are).

Examples:

- Checking balanced expressions
- Recognizing palindromes
(EYE, or RACECAR, or MADAM I'M ADAM)
- Evaluating algebraic expressions

Simple Applications of the ADT Stack: Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
 - An example of balanced braces
$$\text{abc \{ defg\{ ijk \} \l \{ mn \ } op \} qr}$$
 - An example of unbalanced braces
$$\text{abc \{ def \ } \{ ghij \{ kl \ } m}$$

$$\text{abc \{ def \ } \{ ghij \{ kl \ } m}$$

Checking for Balanced Braces

- Requirements for balanced braces
 - Each time you encounter a “}”, it matches an already encountered “{”
 - When you reach the end of the string, you have matched each “{”

Checking for Balanced Braces

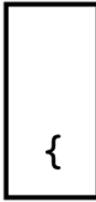
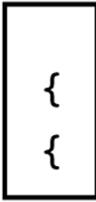
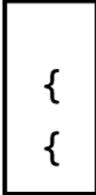
<u>Input string</u>	<u>Stack as algorithm executes</u>			
	1.	2.	3.	4.
{a{b}c}				
	1. push "{"	2. push "{"	3. pop	4. pop
				Stack empty \implies balanced
{a{bc}}				
	1. push "{"	2. push "{"	3. pop	
				Stack not empty \implies not balanced
{ab}c				
	1. push "{"	2. pop		
				Stack empty when last ")" encountered \implies not balanced

Figure 7-3

Traces of the algorithm that checks for balanced braces

Evaluating Postfix Expressions

- A postfix (reverse Polish logic) calculator
 - Requires you to enter postfix expressions
 - Example: 2 3 4 + *
 - Infix = $2 * (3 + 4)$
 - When an operand is entered, the calculator
 - Pushes it onto a stack
 - When an operator is entered, the calculator
 - Applies it to the top two operands of the stack
 - Pops the operands from the stack
 - Pushes the result of the operation on the stack

Evaluating Postfix Expressions

Key entered	Calculator action	Stack (bottom to top)
2	push 2	2
3	push 3	2 3
4	push 4	2 3 4
+	operand2 = pop stack operand1 = pop stack	(4) 2 3 (3) 2
	result = operand1 + operand2 push result	2 2 7
*	operand2 = pop stack operand1 = pop stack	(7) 2 (2)
	result = operand1 * operand2 push result	(14) 14

Figure 7-8

The action of a postfix calculator when evaluating the expression $2 * (3 + 4)$

Evaluate the following Postfix expression

- 3 2 + 4 * 5 1 - /



Exercise

Queue



Bus Stop Queue



Add a people to the queue. – **Put / Enqueue**

Bus Stop Queue



Remove a people from the queue. – **Remove / Dequeue**

Bus Stop Queue



Remove a people from the queue. – **Remove / Dequeue**

Bus Stop Queue



front

rear



Add a people to the queue. – **Put / Enqueue**

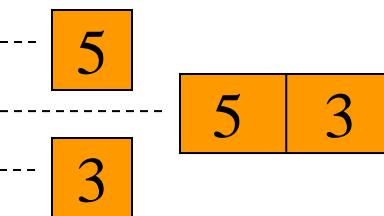
Remove a people from the queue. – **Remove / Dequeue**

A queue is a **FIFO** list. (**First In First Out**)

Queue operations

This table shows a series of queue operations and their effects. The queue is empty initially.

Operation	Output	$\text{front} < - Q \quad <- \text{rear}$
enqueue(5)	-	(5)
enqueue(3)	-	(5,3)
dequeue()	5	(3)
enqueue(7)	-	(3,7)
dequeue()	3	(7)
front()	7	(7)
dequeue()	7	()
dequeue()	“error”	()
isEmpty()	true	()
enqueue(9)	-	(9)
enqueue(7)	-	(9,7)
size()	2	(9,7)
enqueue(3)	-	(9,7,3)
enqueue(5)	-	(9,7,3,5)
dequeue()	9	(7,3,5)



Operation of Queue

isEmpty();

getFrontElement();

getRearElement();

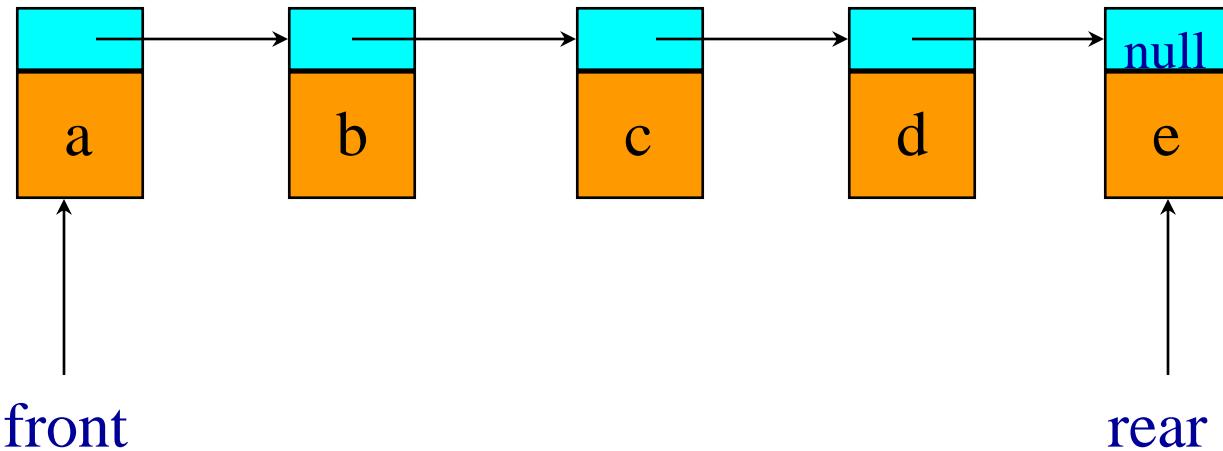
put(Object theObject) #enqueue

remove() #dequeue

LinkedQueue.py

Linked List Implementation of Queue

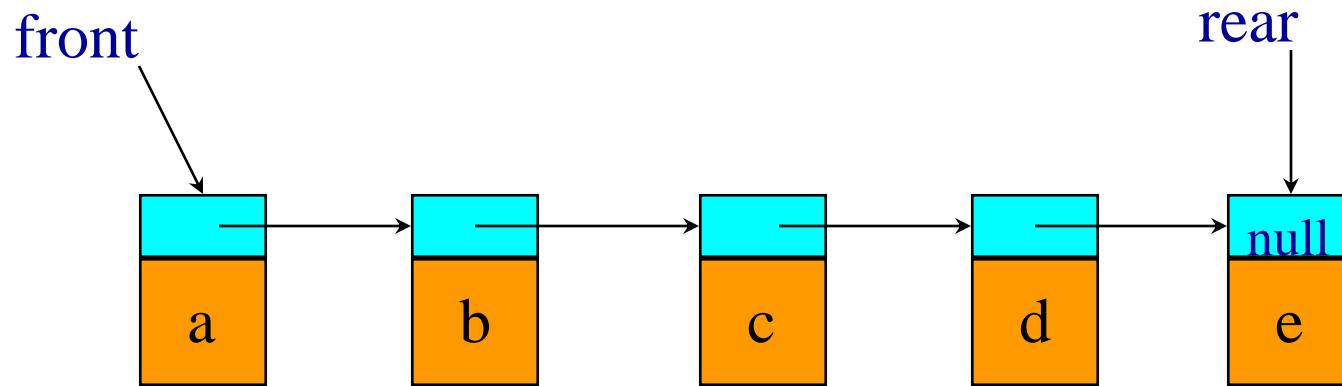
LinkedQueue



- when front is left end of list and rear is right end

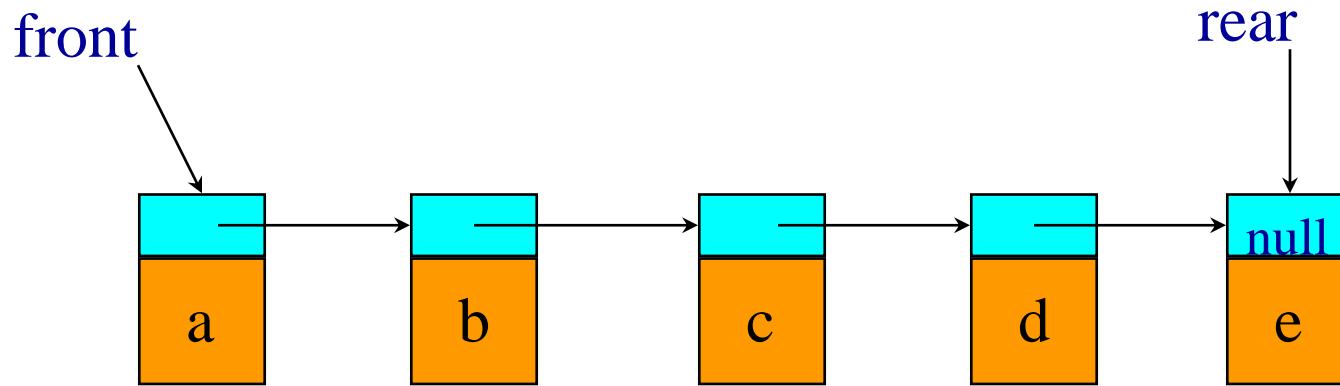
LinkedQueue.py

LinkedQueue



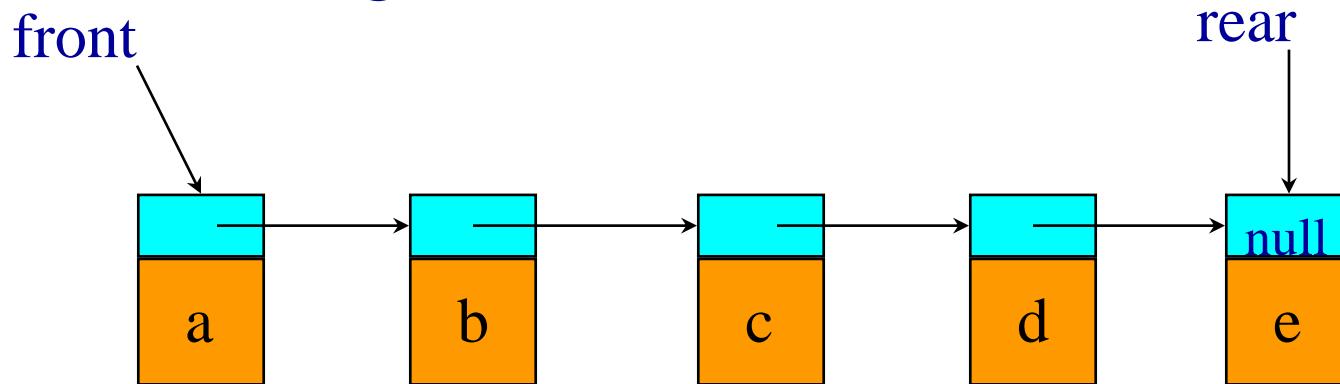
```
class Node:  
    def __init__(self, el = None, n = None):  
        self.next = n  
        self.element = el  
  
class LinkedQueue:  
    def __init__(self):  
        self.front = None  
        self.rear = None  
        self.size = 0
```

isEmpty()



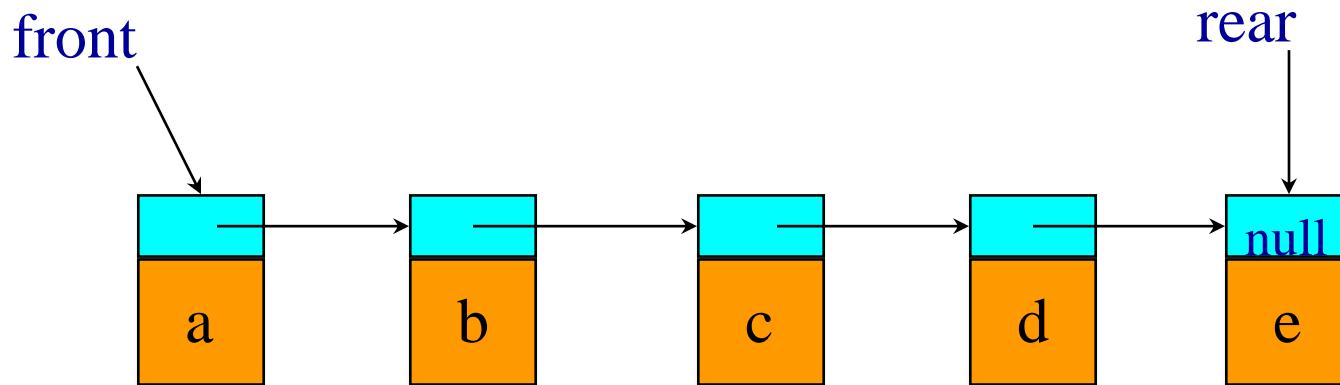
```
#return true if the stack is empty
def isEmpty(self):
    return self.front == None
```

getFrontElement()



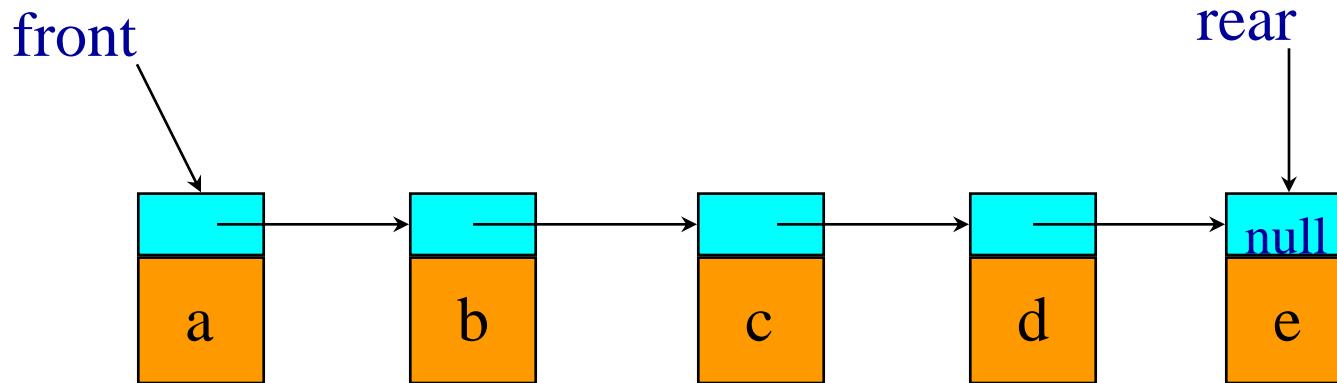
```
#return the first element
def getFrontElement(self):
    if (self.isEmpty()):
        return None;
    else:
        return self.front.element;
```

getRearElement()



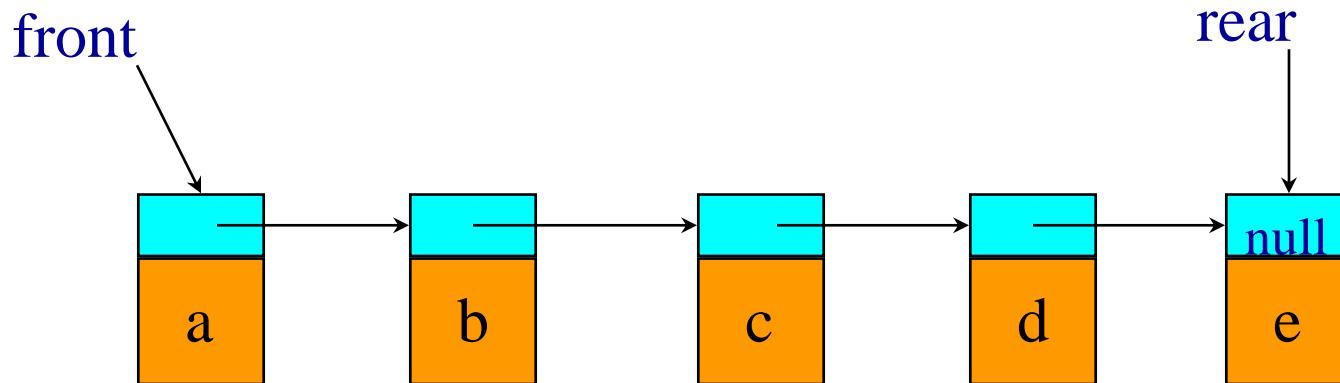
```
#return the last element
def getRearElement(self):
    if (self.isEmpty()):
        return None;
    else:
        return self.rear.element;
```

put(Object theElement)



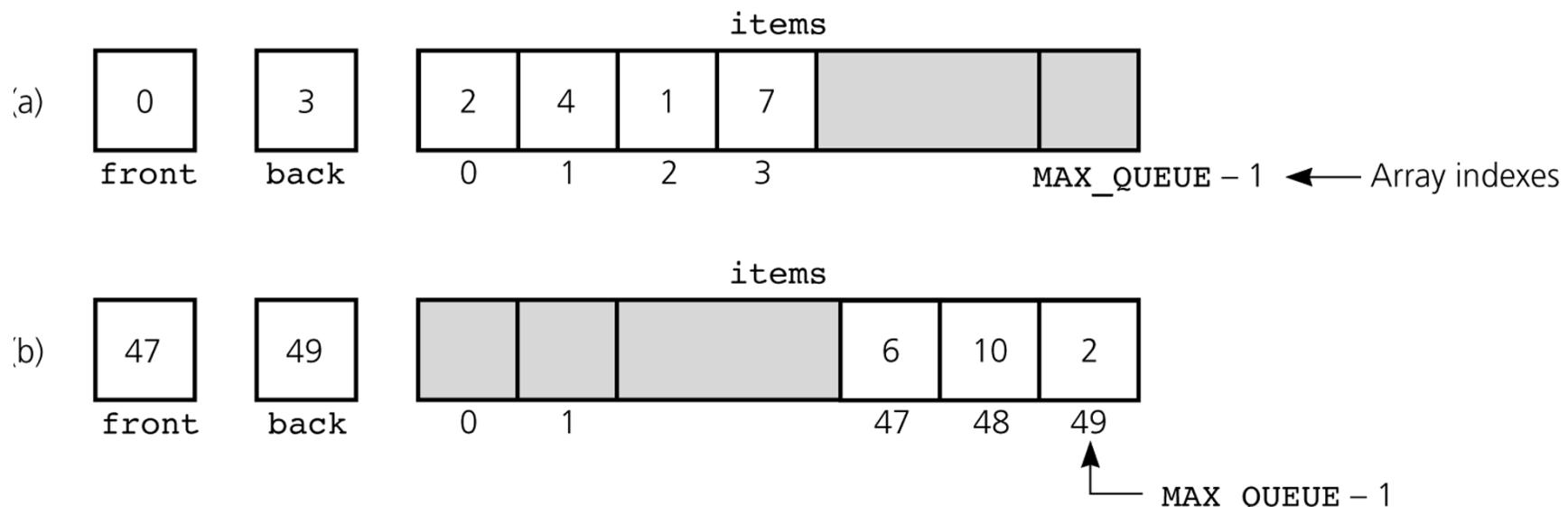
```
#add an element in the queue/enqueue
def put(self, element):
    p = Node(element)
    if(self.isEmpty()):
        self.front = p      #empty queue
    else:
        self.rear.next = p      #nonempty queue
    self.rear = p
```

remove()



```
#remove an element in the queue/dequeue
def remove(self):
    if(self.isEmpty()):
        return None
    frontElement = self.front.element
    self.front = self.front.next
    if(self.isEmpty()):
        self.rear = None
    return frontElement
```

An Array-Based Implementation



- a) A naive array-based implementation of a queue; b) rightward drift can cause the queue to appear full

Summary

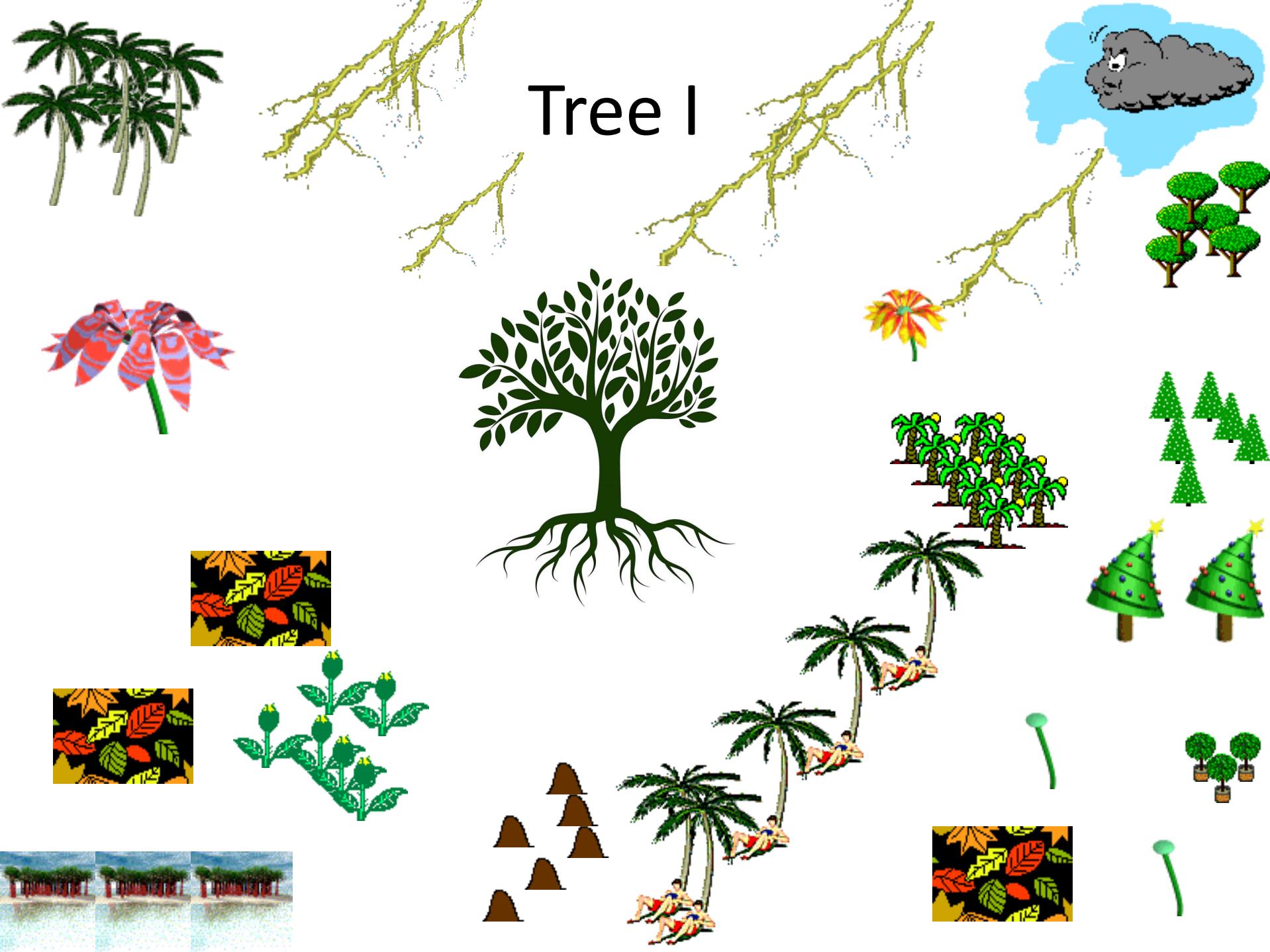
- Stacks
 - Linear list.
 - One end is called top.
 - Other end is called bottom.
 - Additions to and removals from the top end only.
- Queue
 - Linear list.
 - One end is called front.
 - Other end is called rear.
 - Additions are done at the rear only.
 - Removals are made from the front only.

SEHH2239

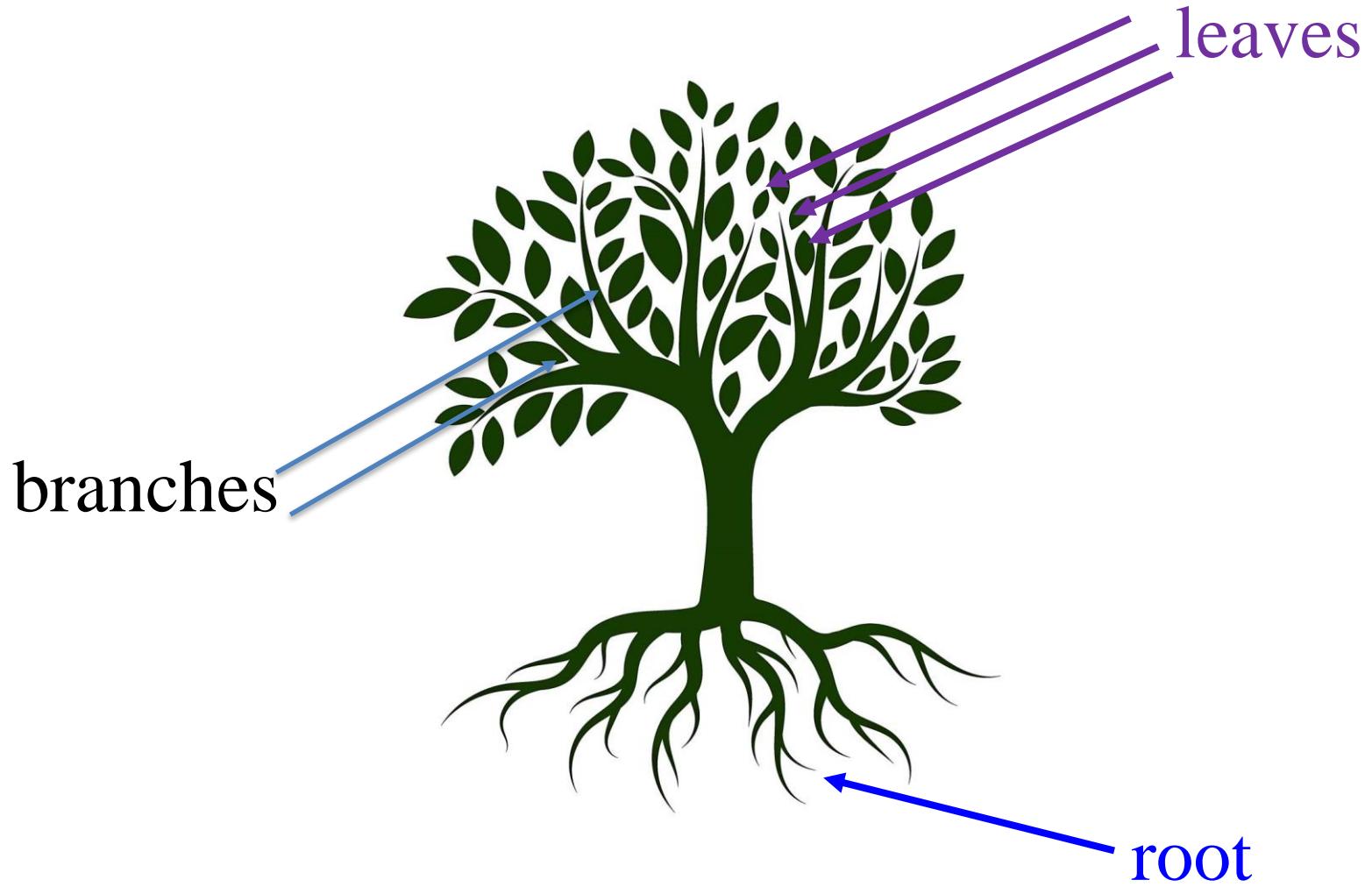
Data Structures

Lecture 8

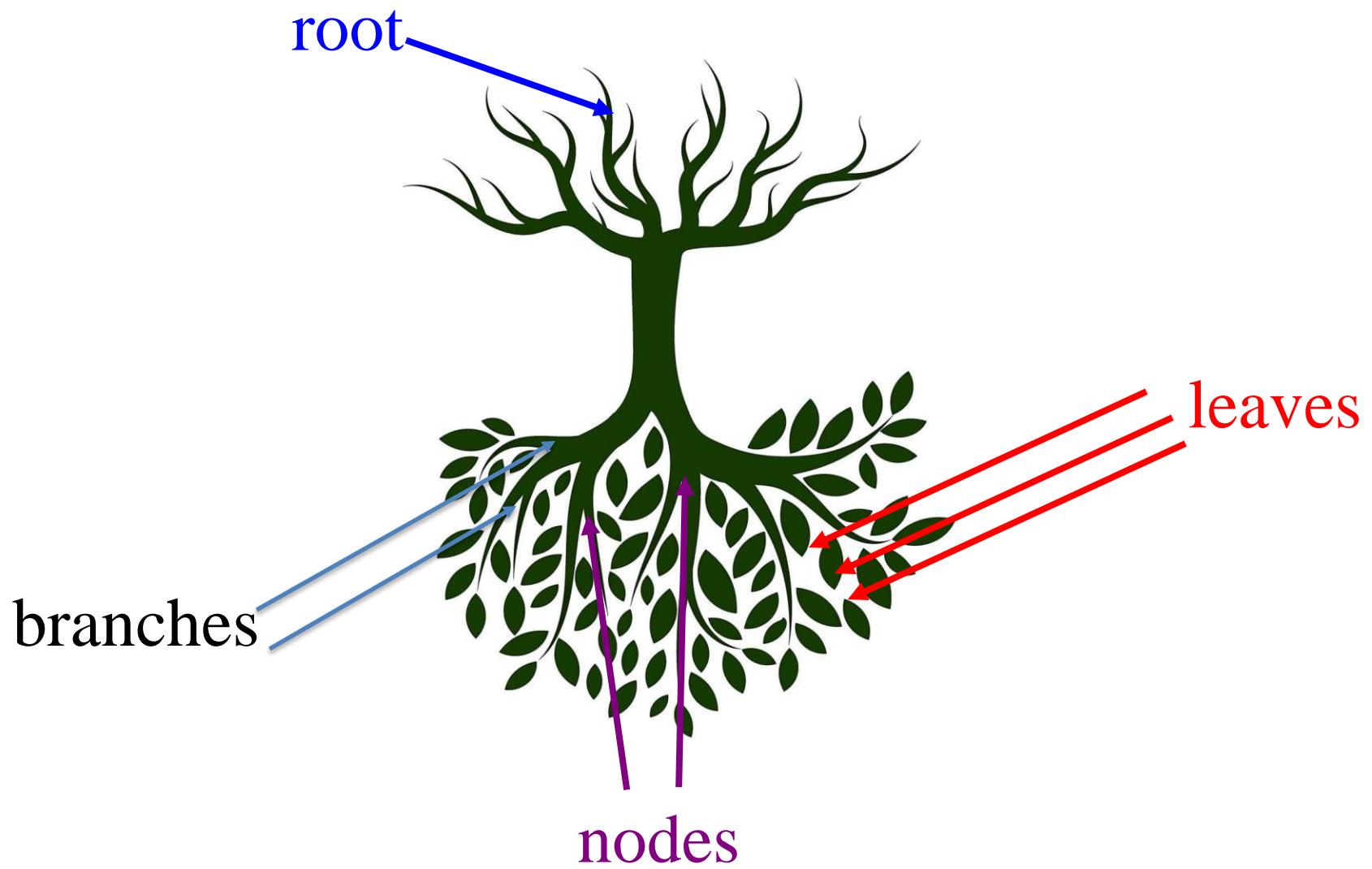
Tree I



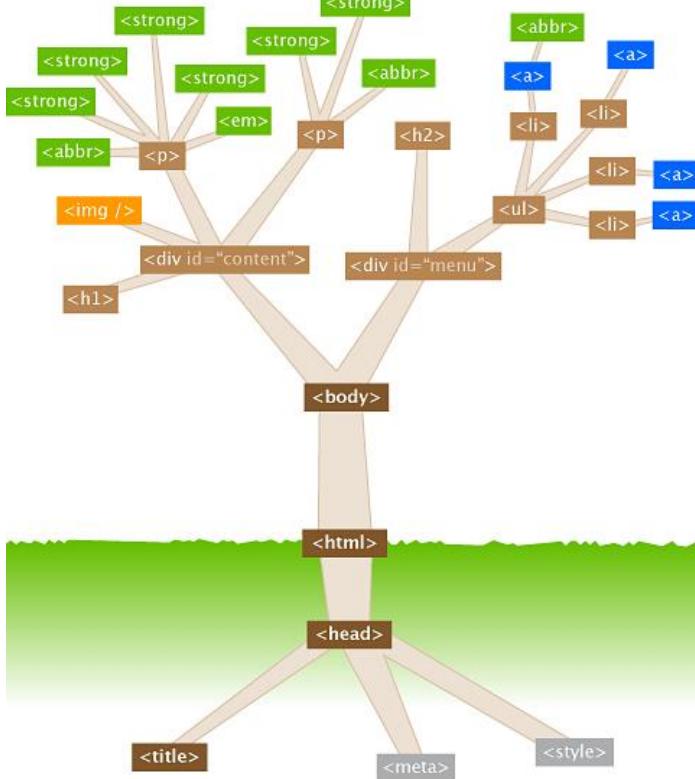
Nature Lover's View Of A Tree



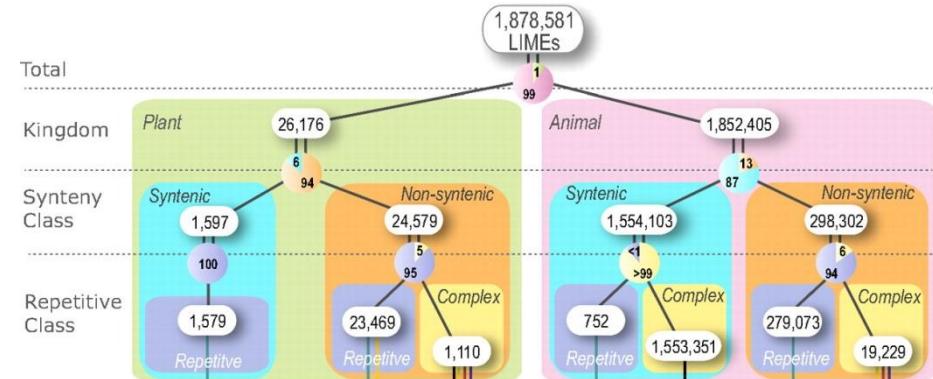
Computer Scientist's View



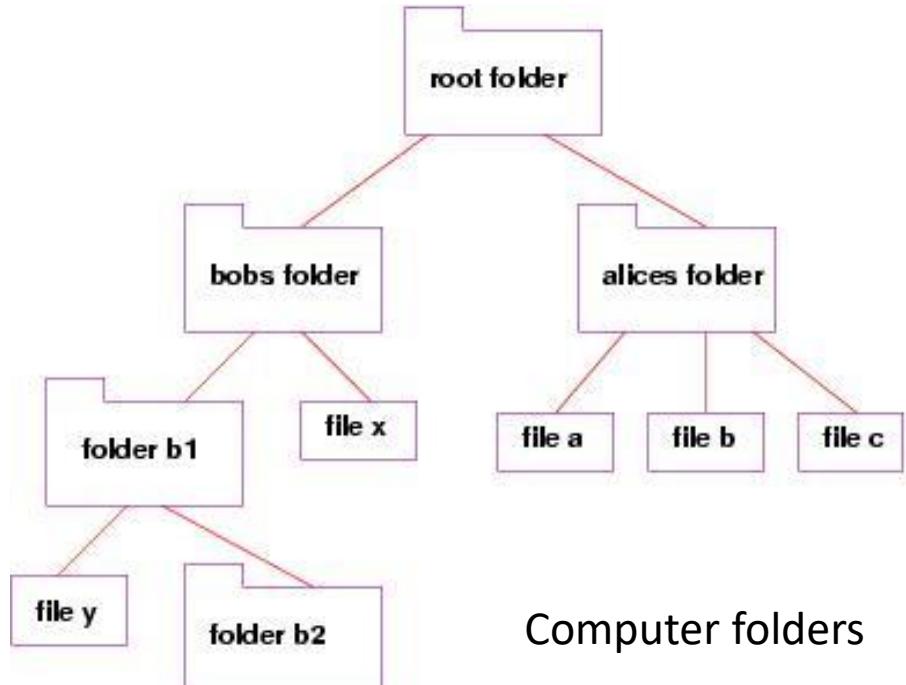
Example of Trees



HTML Document Structure



Organism Classifications

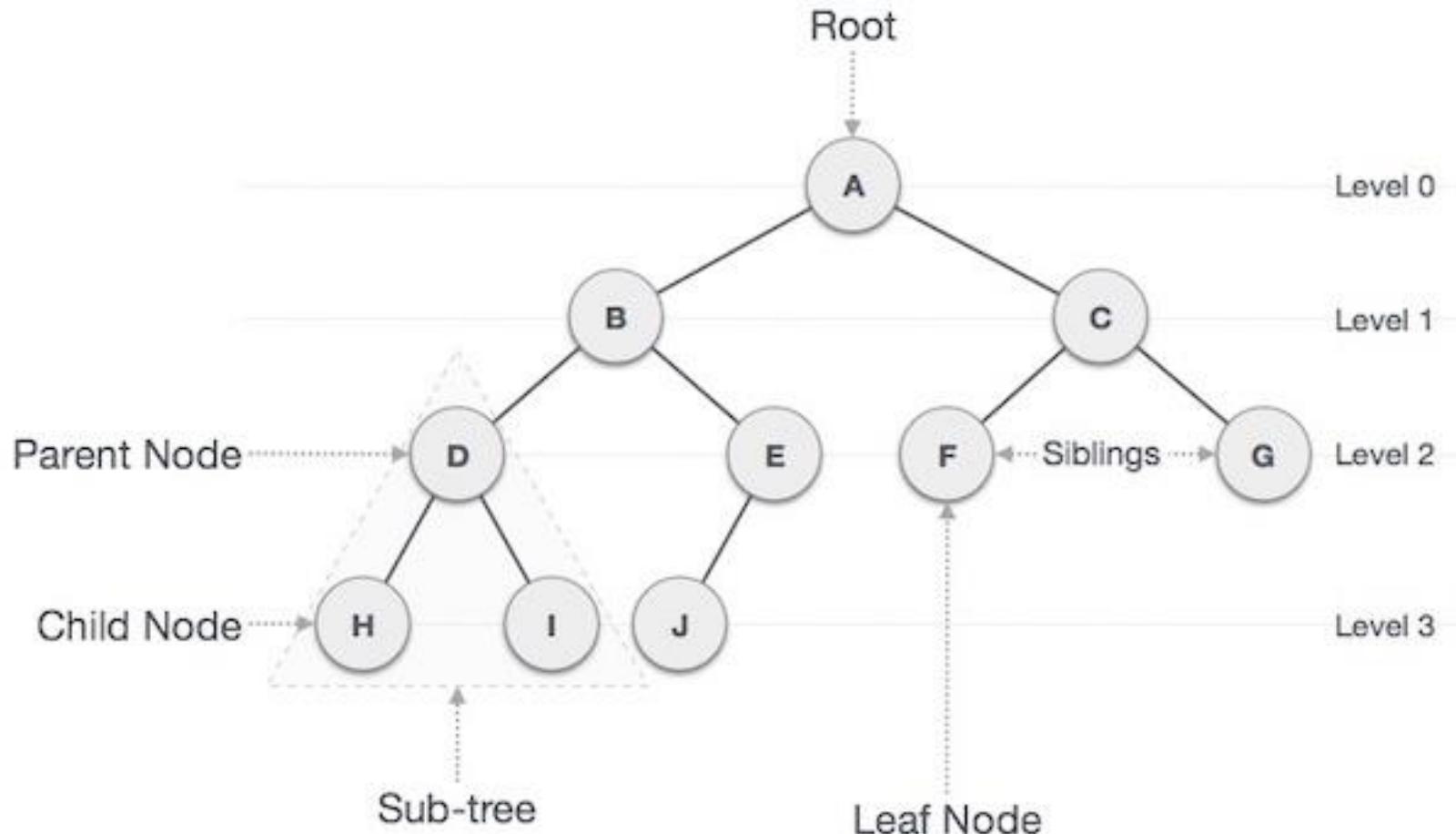


Computer folders

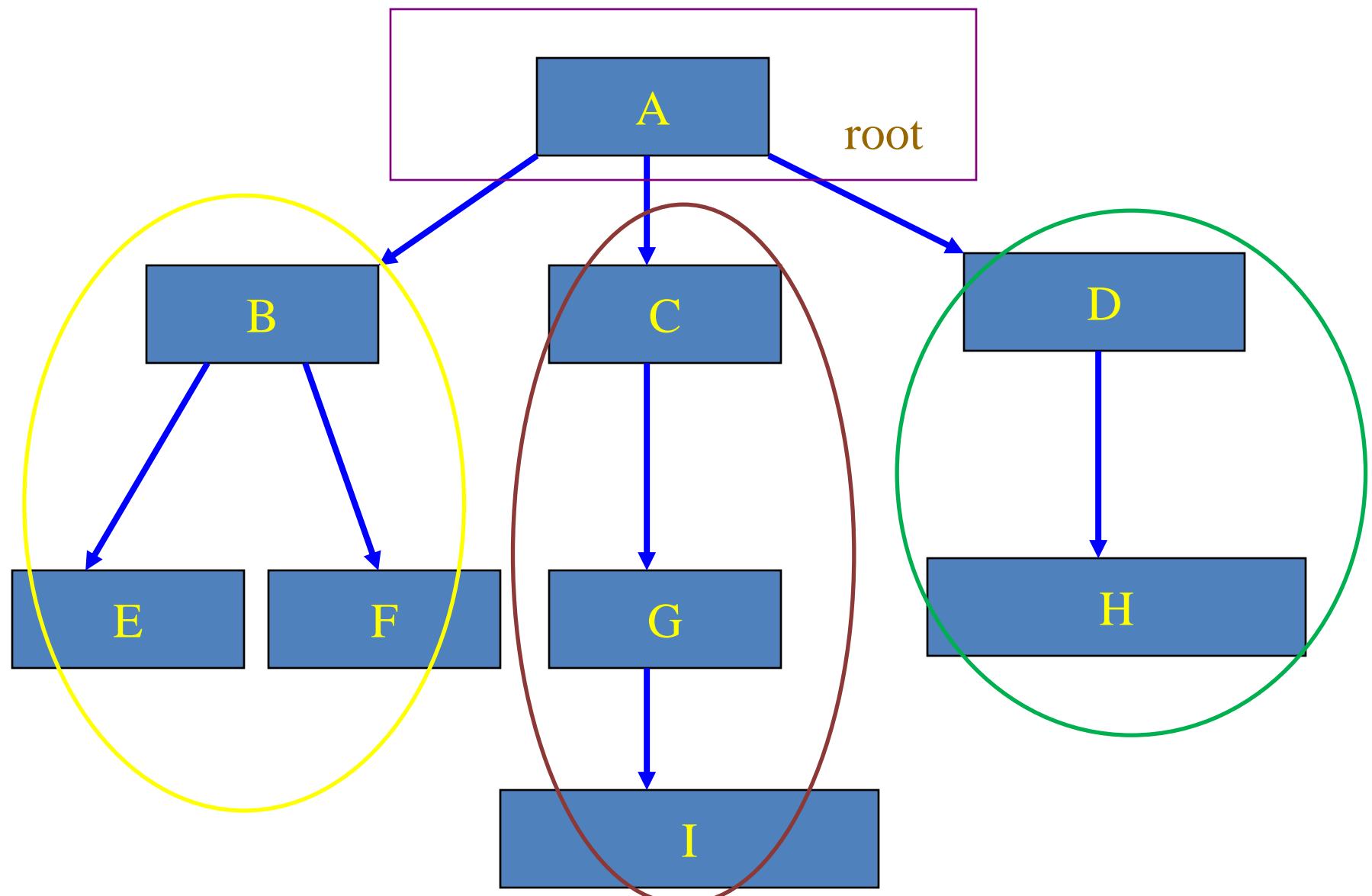
Definition

- A tree t is a finite nonempty set of elements.
- One of these elements is called the root.
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of t .

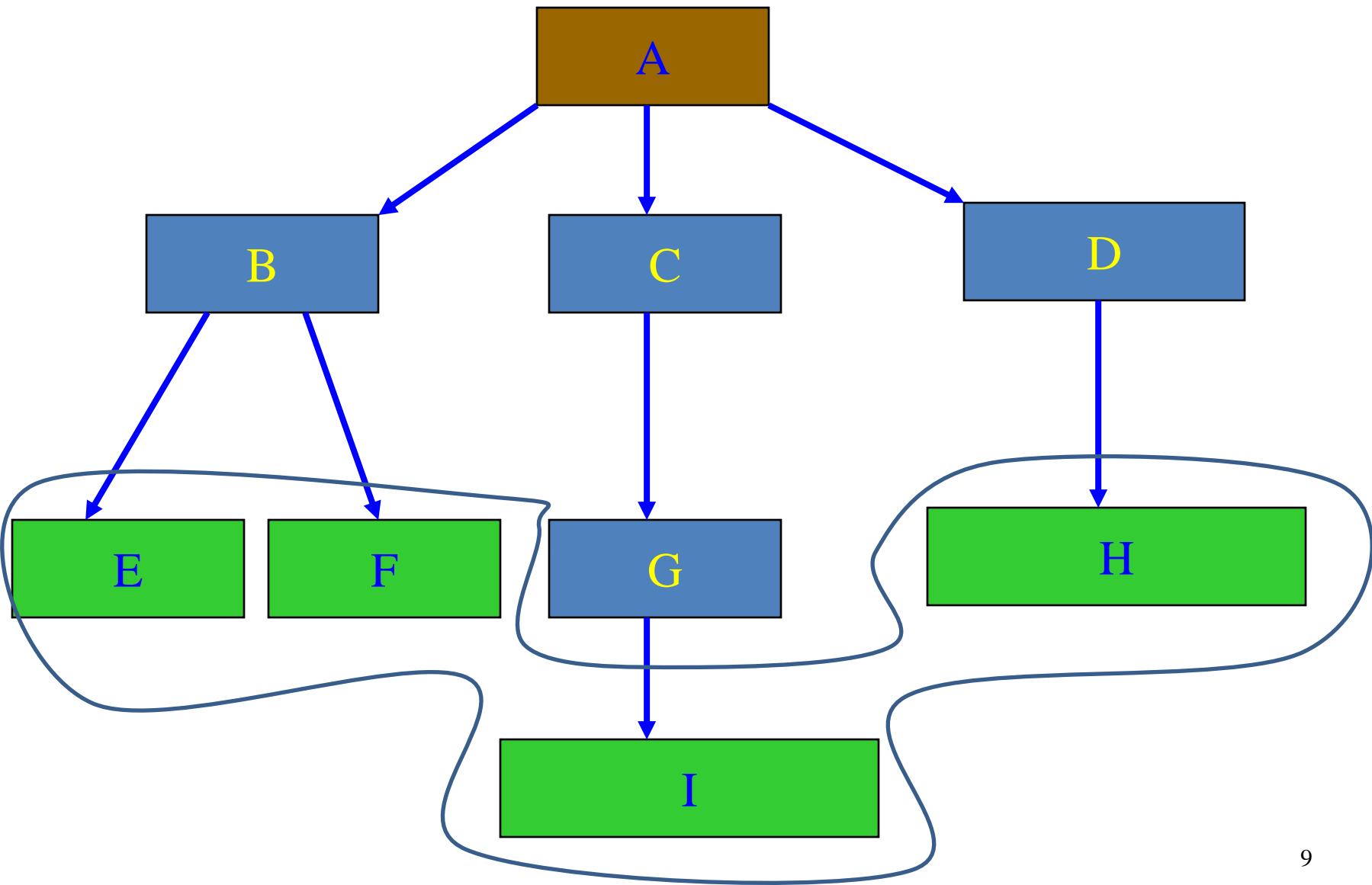
Tree



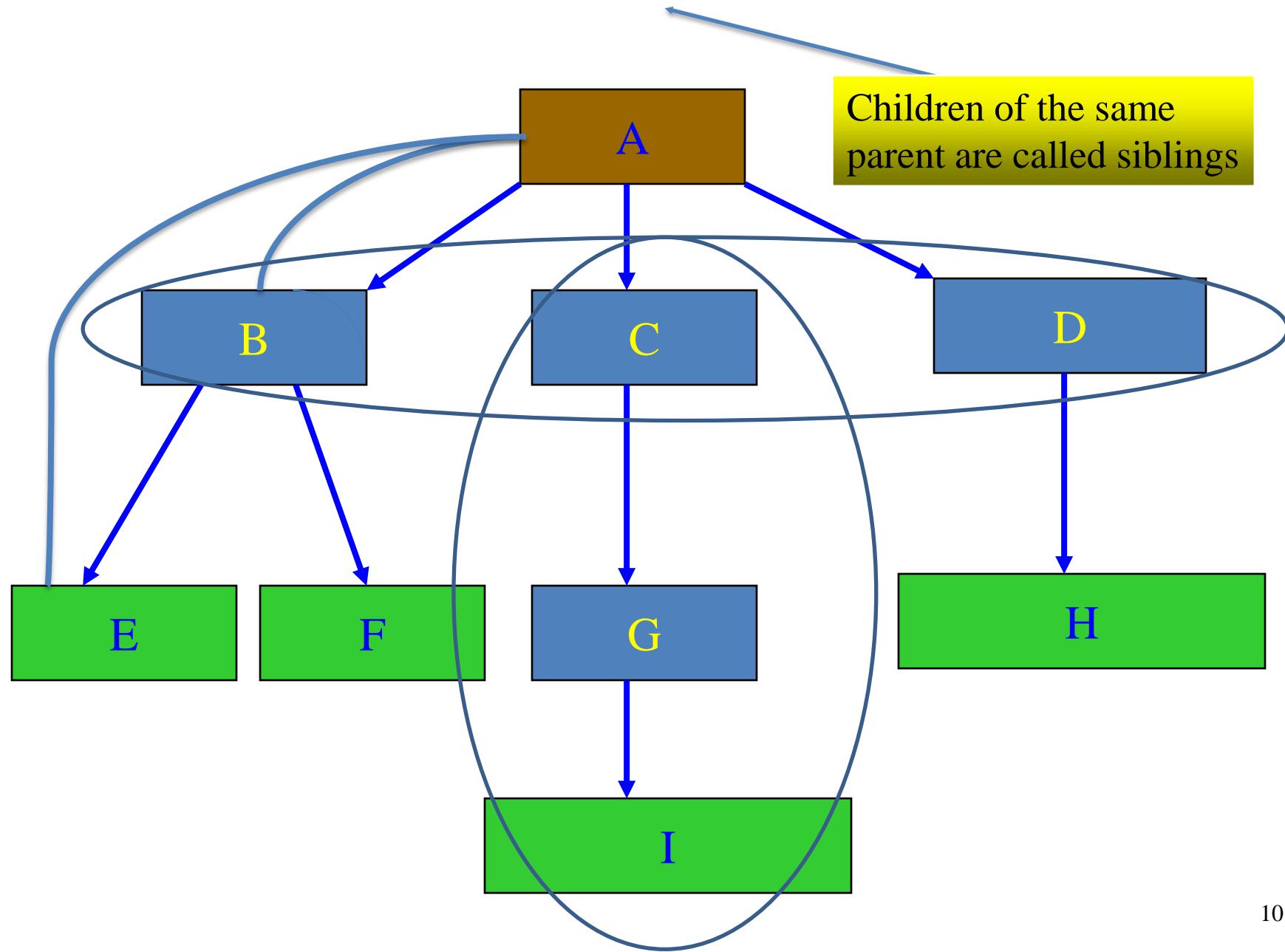
Subtrees



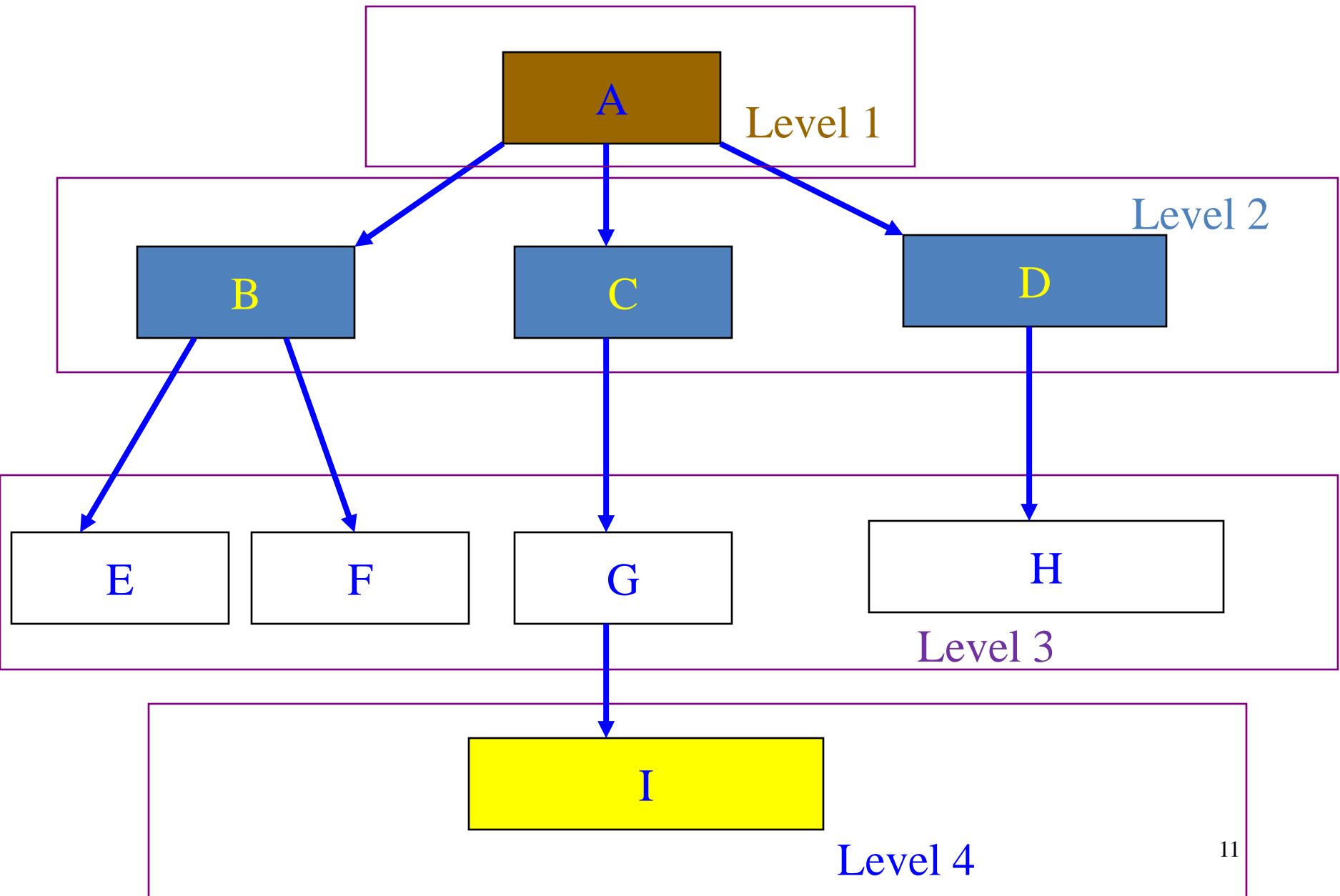
Leaves



Parent, Grandparent, Siblings, Ancestors, Descendants



Levels





Caution

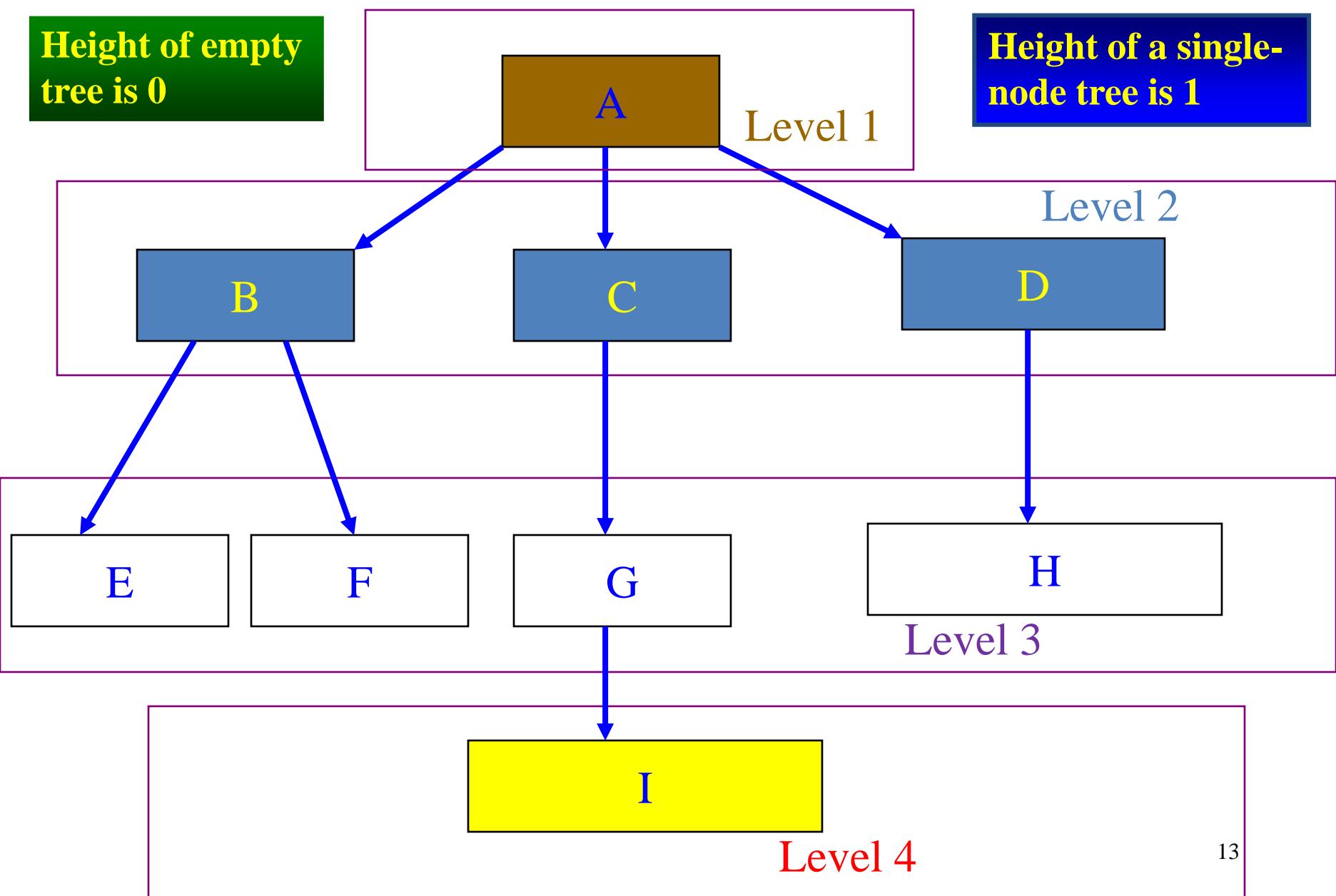


- Some texts start level numbers at 0 rather than at 1 (e.g. The textbook by Hubbard and Huray)
- We shall number levels as follows:
- Root is at level 1.
- Its children are at level 2.
- The grand children of the root are at level 3.
- And so on.

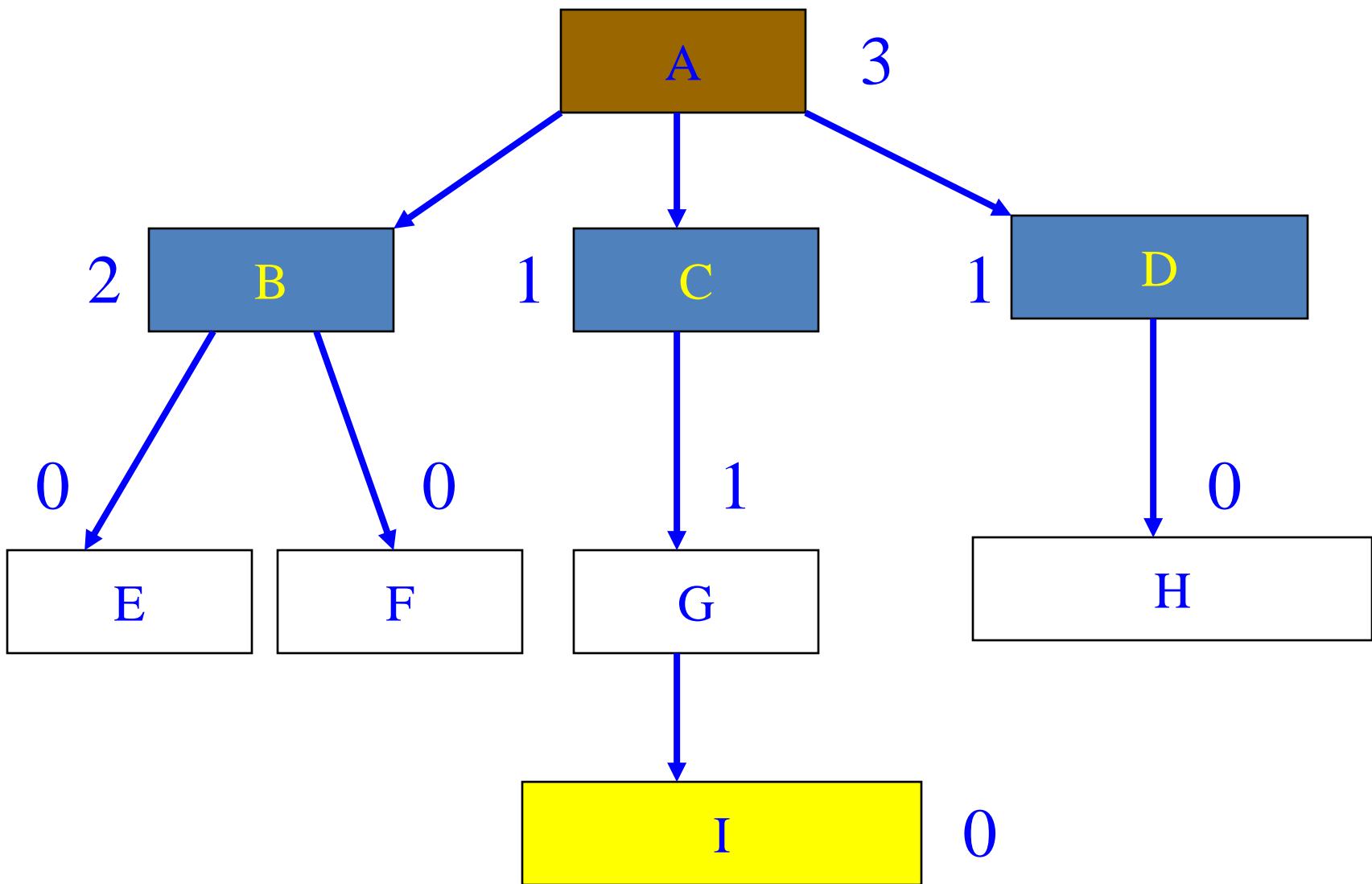
height = depth = highest level number

Height of empty tree is 0

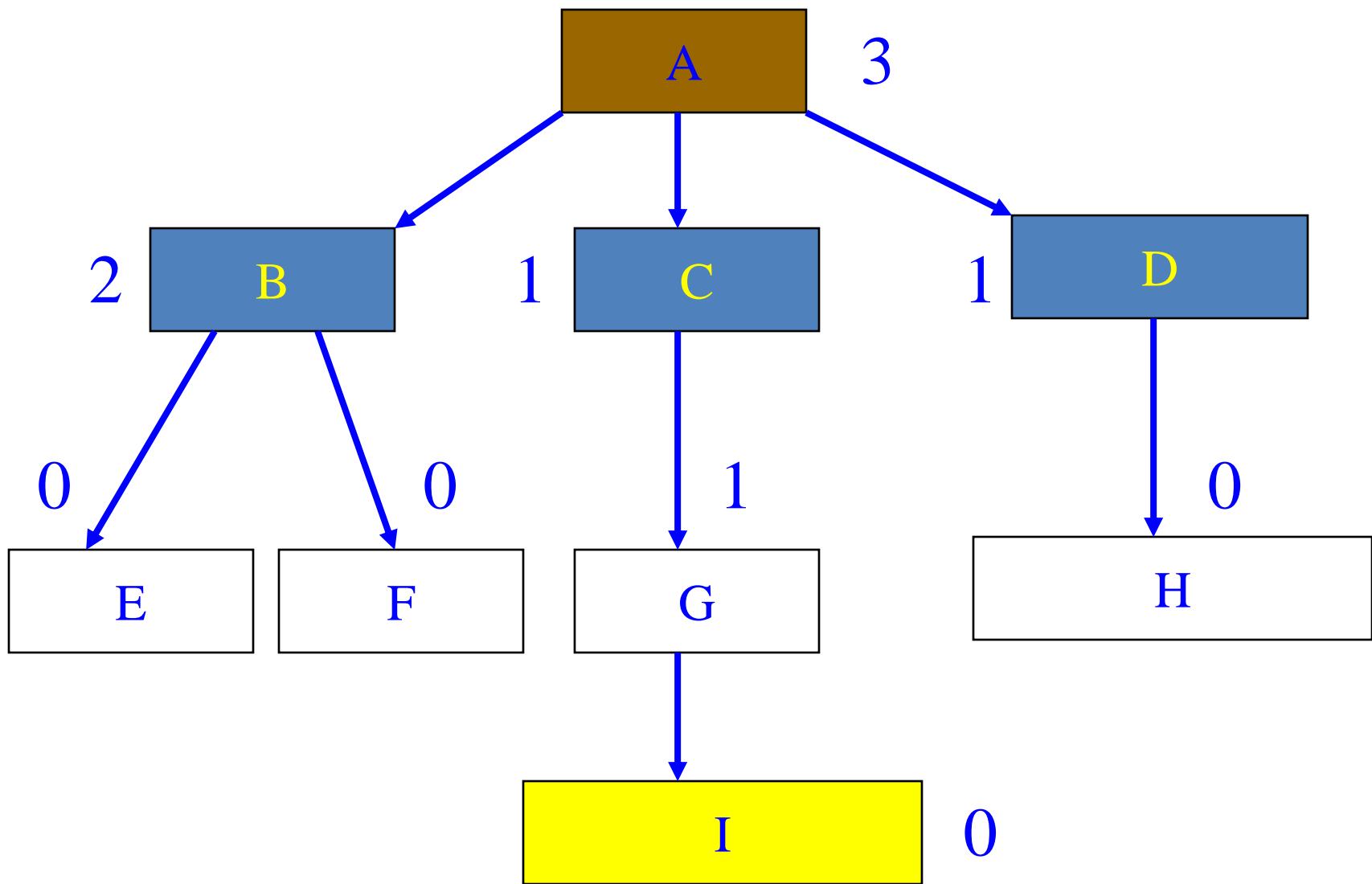
Height of a single-node tree is 1



Node Degree = Number Of Children



Tree Degree = Max Node Degree

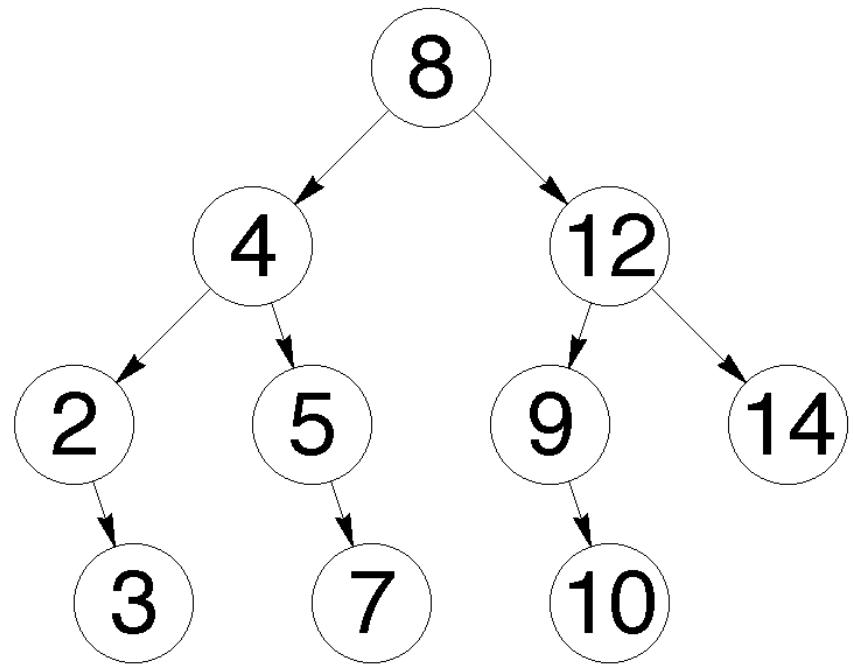


Degree of tree = 3.

Tree - Exercise

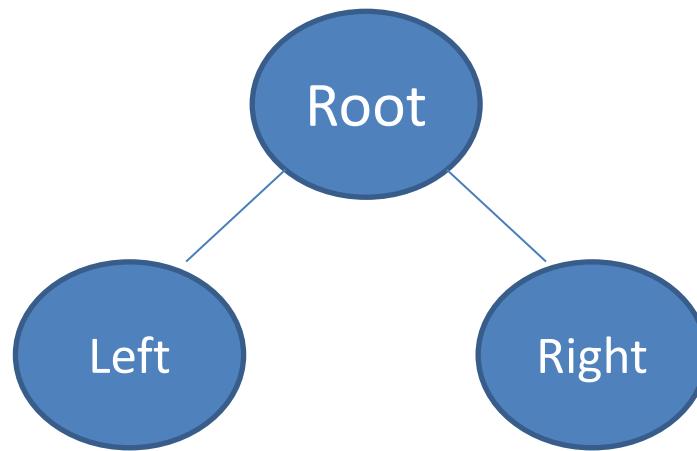
- Find the followings:

1. Root
2. Leave
3. Siblings
4. Height
5. Depth
6. Tree Degree



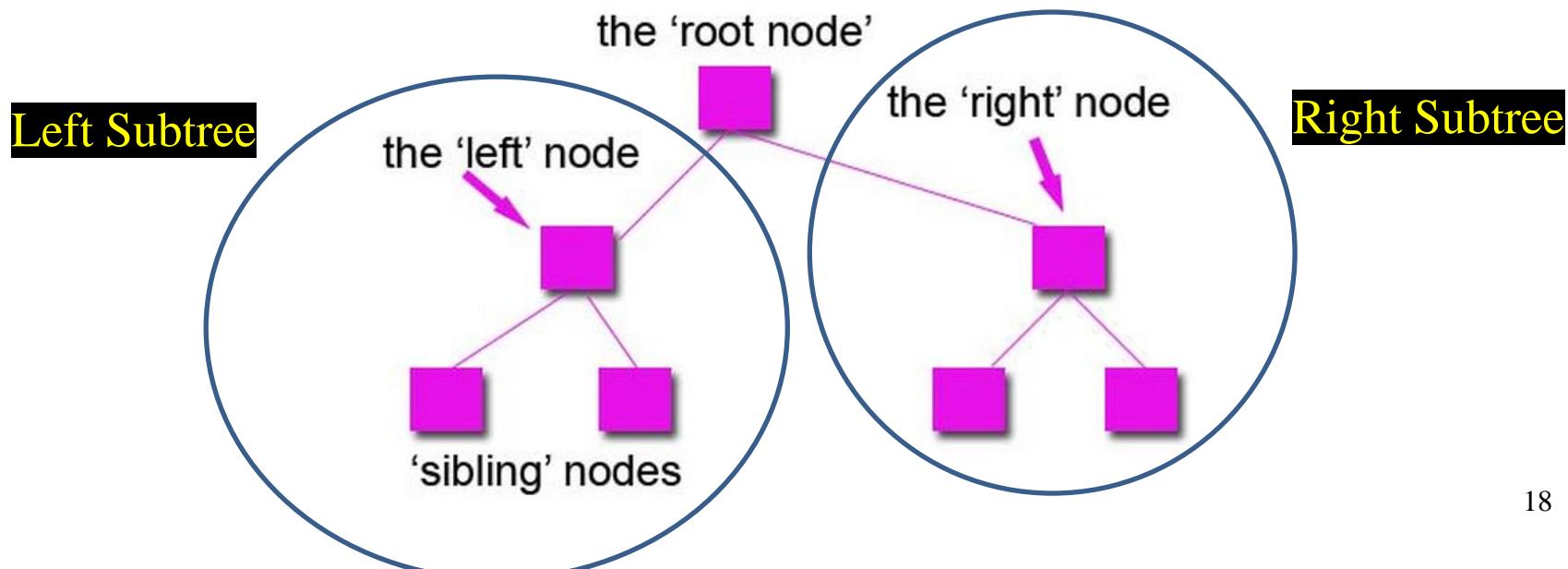
Binary Tree

- Finite (possibly empty) collection of elements.
- A **nonempty** binary tree has a **root** element.
- The remaining elements (if any) are partitioned into ***two binary trees***.
- These are called the **left** and **right** subtrees of the binary tree.



Differences Between A Tree & A Binary Tree

- **Binary tree:**
 - No node may have a degree more than **2**.
- **Tree:**
 - **No limit** on the degree of a node in a tree.



Differences Between A Tree & A Binary Tree

- The subtrees of a **binary tree** are **ordered**; those of a tree are not ordered.



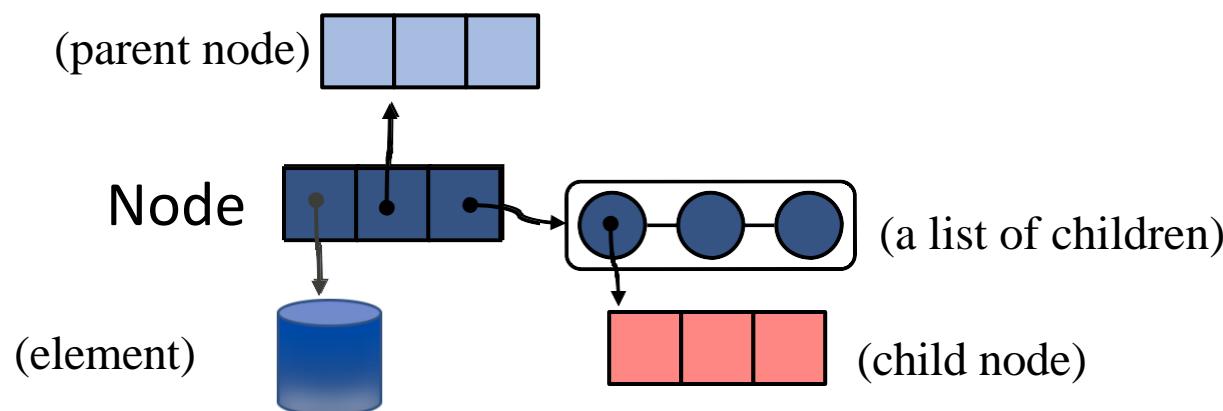
- Are different when viewed as binary trees.
- Are the same when viewed as trees.

Tree Implementation

Structure for a Tree

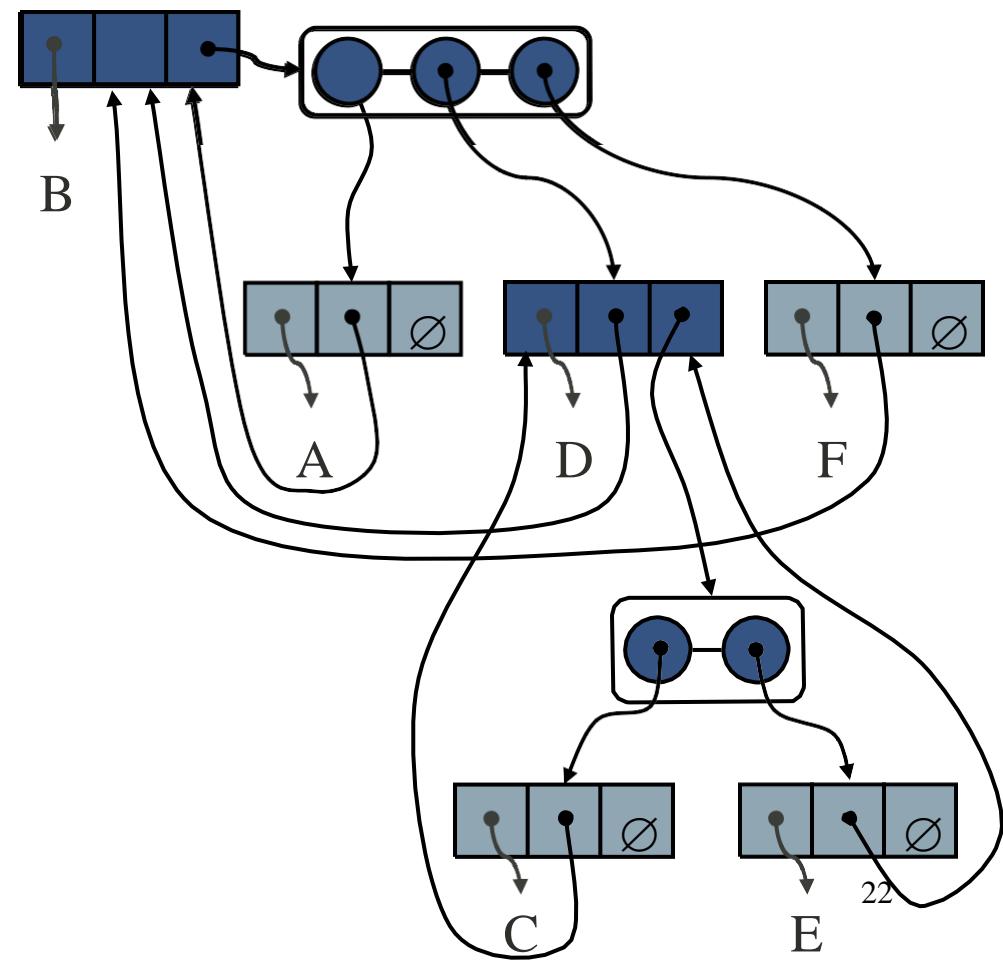
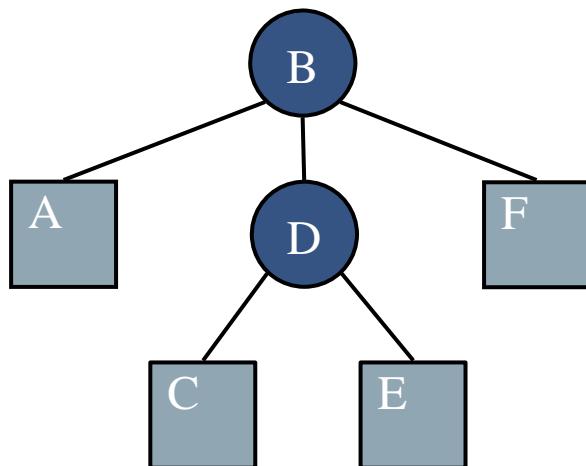
A **Node** is represented by an object
Data variables:

- Element
- A parent node
- A sequence of children nodes



Structure for a Tree

Try to implement the following Tree:

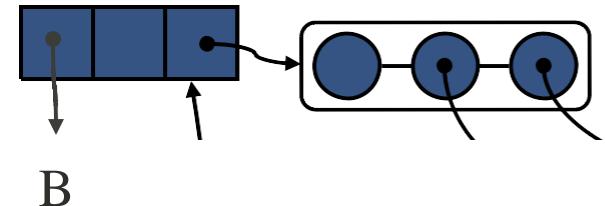
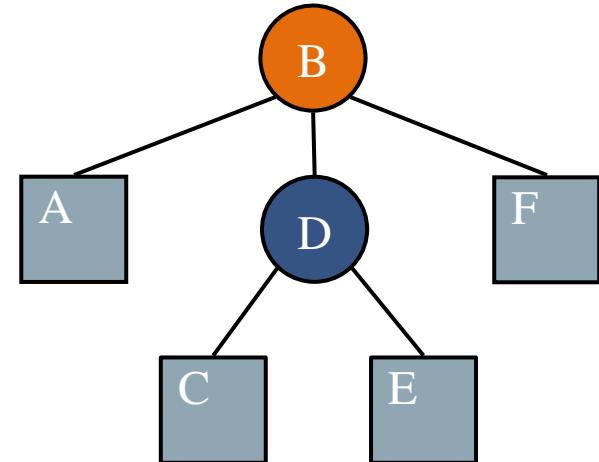


Structure for a Tree

- Create the root

```
class Tree:  
    def __init__(self, data):  
        self.children = []  
        self.data = data
```

```
root = Tree("B")
```



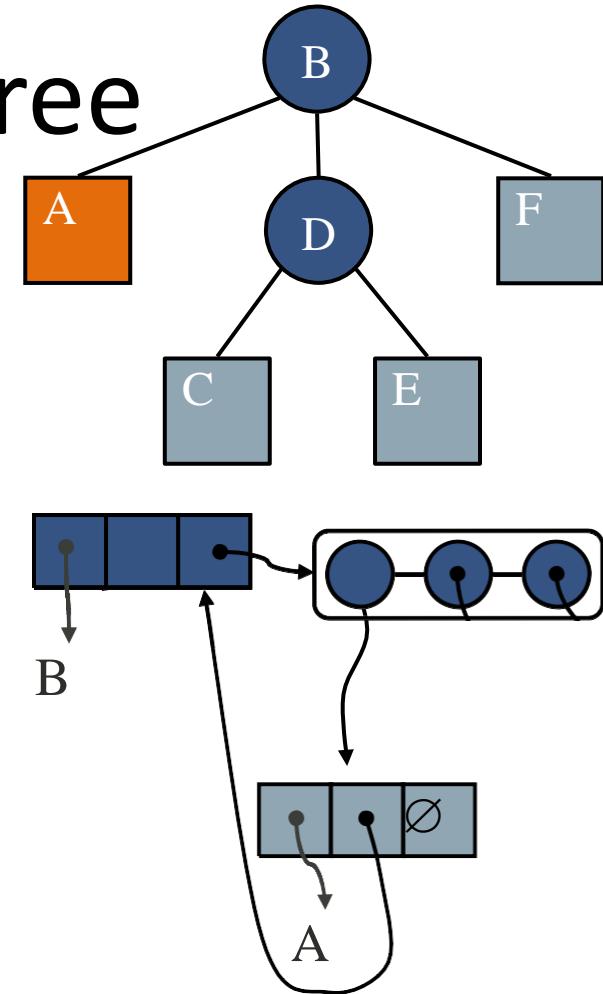
Tree.py

Structure for a Tree

- Add a child

```
class Tree:  
    def __init__(self, data):  
        self.children = []  
        self.data = data
```

```
left = Tree("A")  
middle = Tree("D")  
right = Tree("F")  
root = Tree("B")  
root.children = [left, middle, right]
```



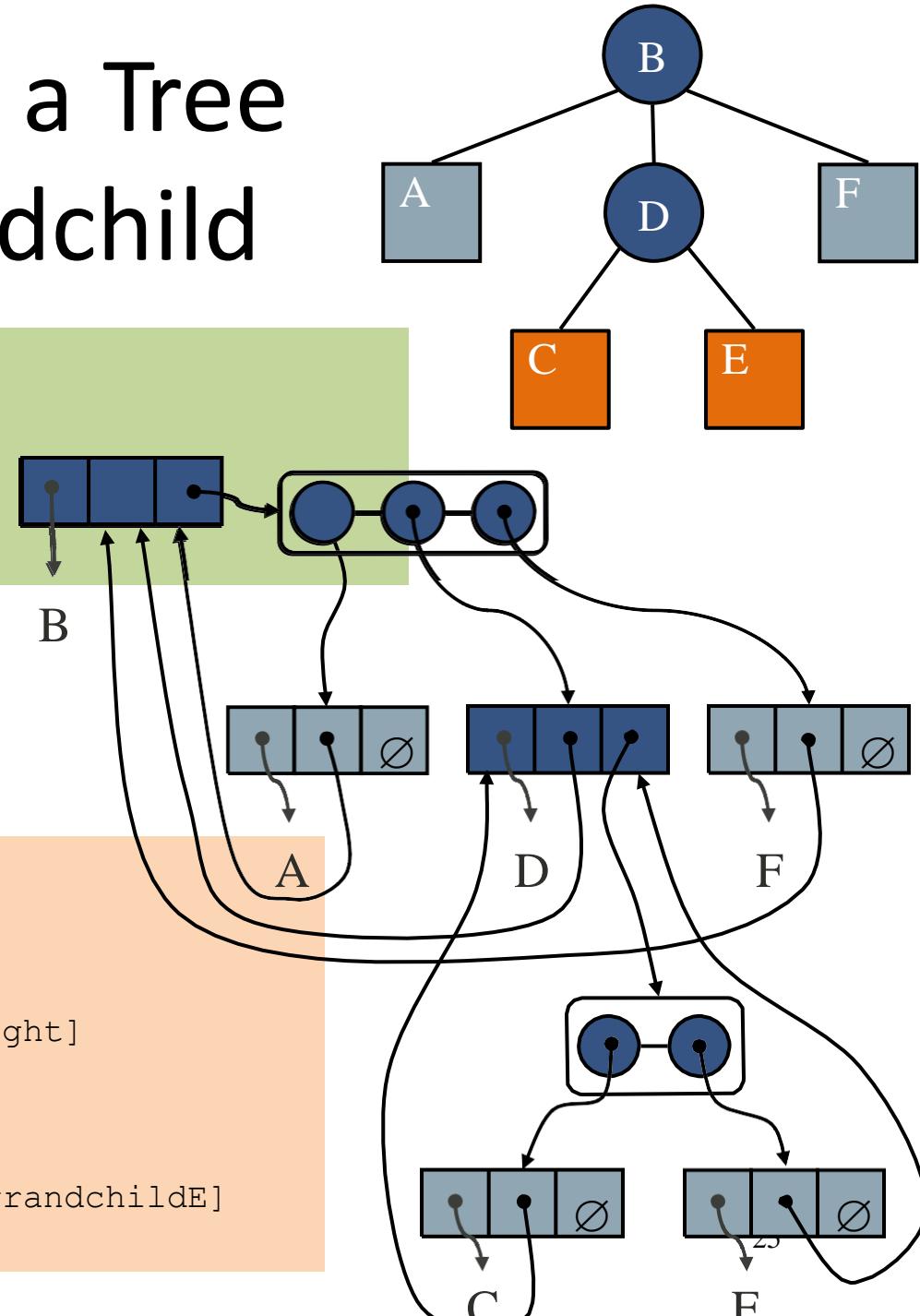
Tree.py

Structure for a Tree

- Add a grandchild

```
class Tree:  
    def __init__(self, data):  
        self.children = []  
        self.data = data
```

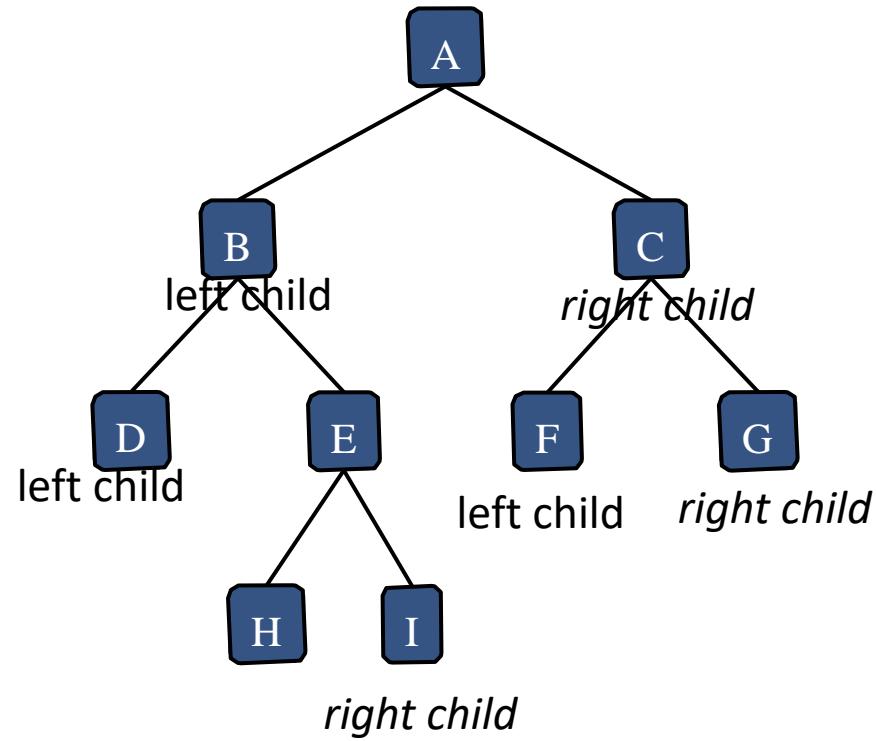
```
left = Tree("A")  
middle = Tree("D")  
right = Tree("F")  
root = Tree("B")  
root.children = [left, middle, right]  
  
grandchildC = Tree("C")  
grandchildE = Tree("E")  
middle.children = [grandchildC, grandchildE]
```



Binary Tree ADT and Implementation in Linked Nodes

Binary Tree

- A binary tree is a tree with the following properties:
 1. Each *internal* node has *at most two* children
 2. The children of a node are an *ordered pair* (*left* and *right*)
 3. We call the children of an internal node *left child* and *right child*

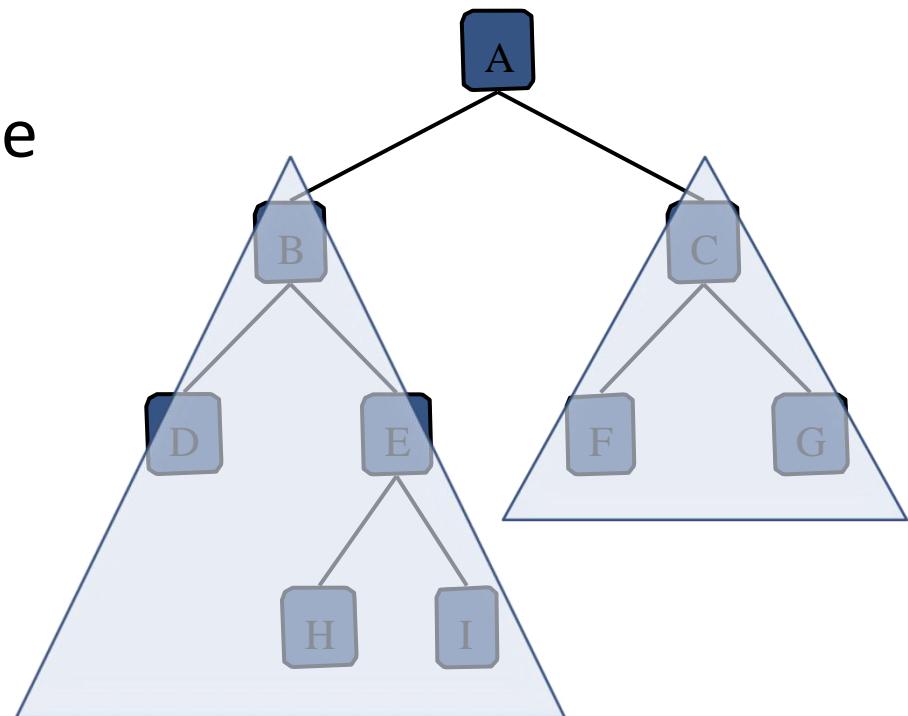


Binary Tree

Alternative *recursive definition*:

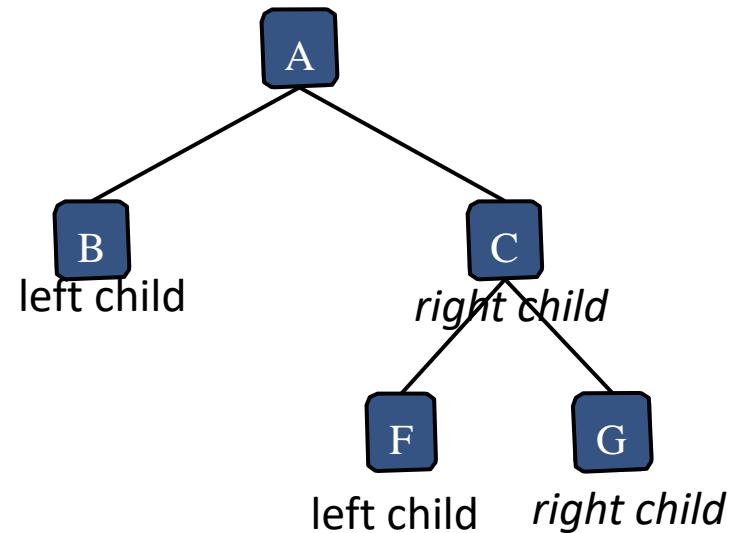
a binary tree is either

- a tree consisting of a single node, or
- a tree whose root has an ***ordered pair of children***, each of which is a binary tree



BinaryTree Node

```
class BinaryTreeNode:  
    def __init__(self, data):  
        self.left = None  
        self.right = None  
        self.data = data  
  
a = BinaryTreeNode("A")
```

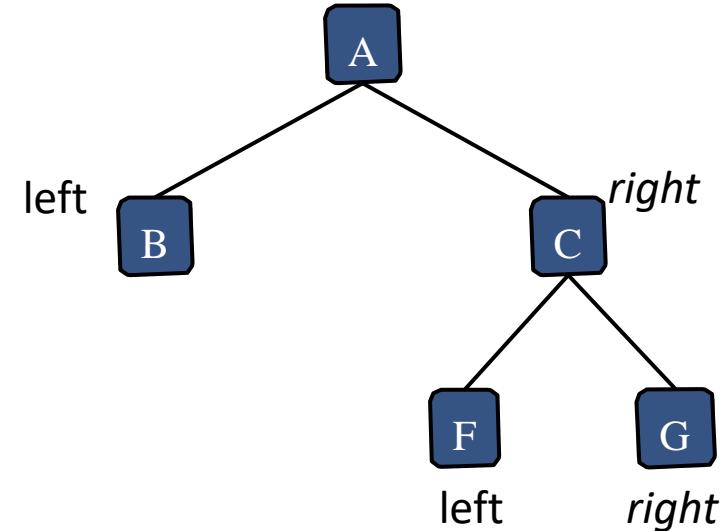


BinaryTree.py

BinaryTree Node

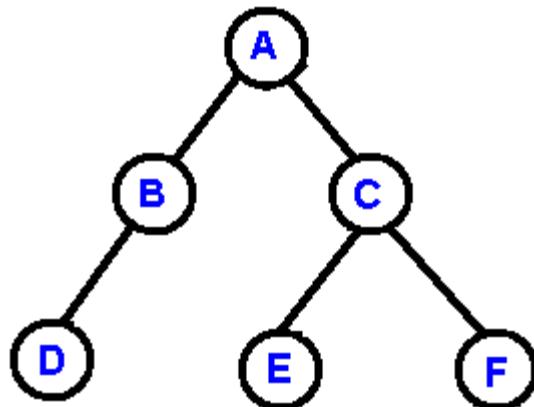
```
# Create Binary Tree
a = BinaryTreeNode("A")
b = BinaryTreeNode("B")
c = BinaryTreeNode("C")
a.setLeftChild(b)
a.setRightChild(c)

f = BinaryTreeNode("F")
g = BinaryTreeNode("G")
c.setLeftChild(f)
c.setRightChild(g)
```



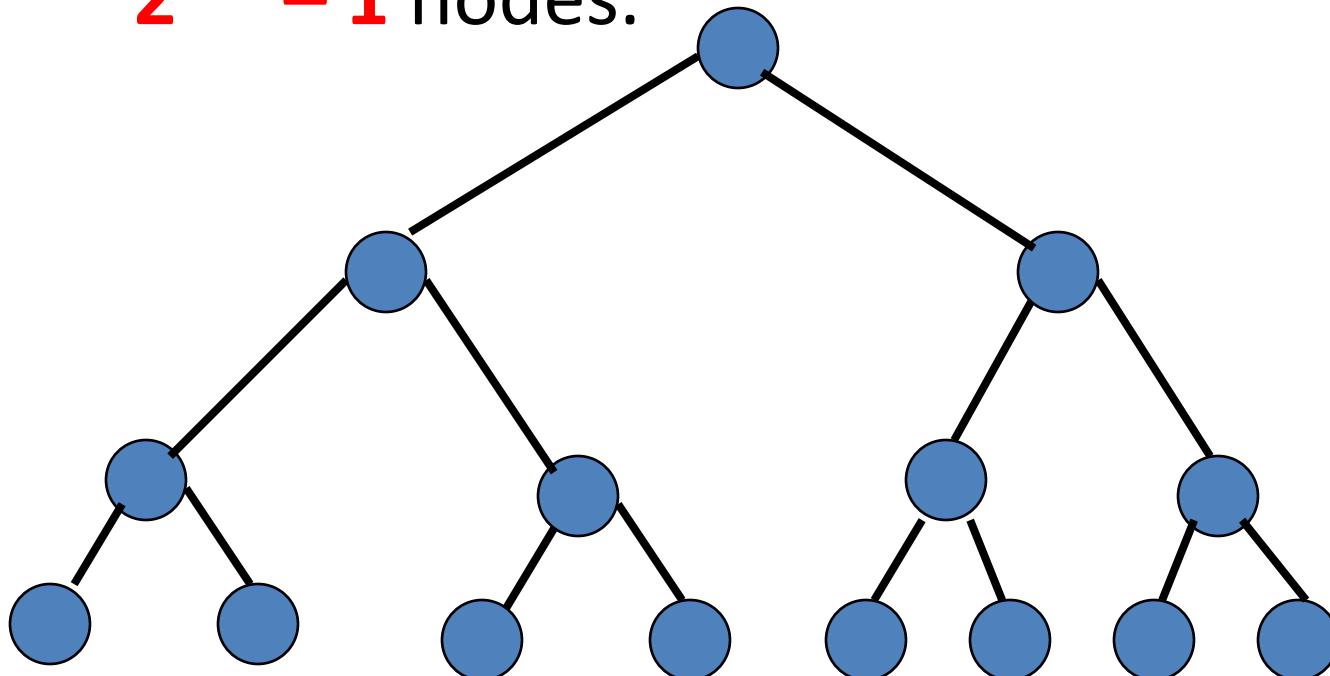
```
def setLeftChild(self, theLeftChild):
    left = theLeftChild
def setRightChild(self, theRightChild):
    right = theRightChild
```

Special Binary Trees



Properties - Full Binary Tree

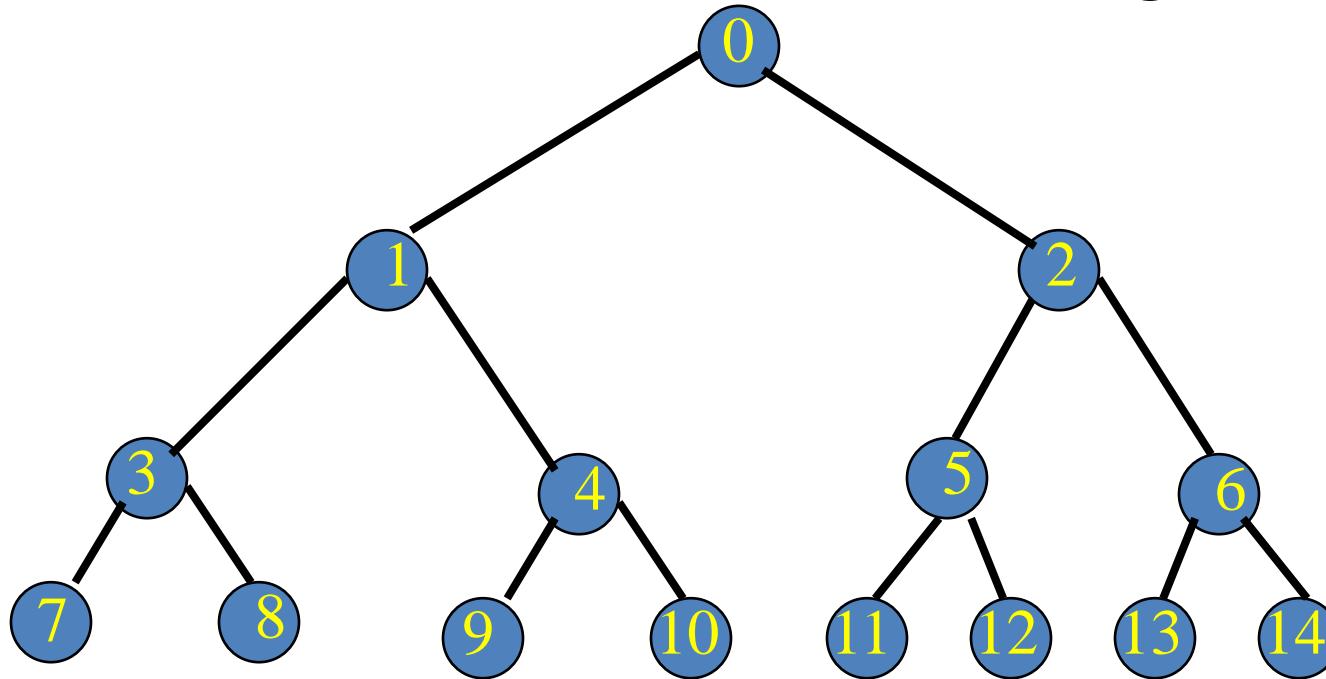
- A full binary tree of a given height \mathbf{h} has $2^{\mathbf{h+1}} - 1$ nodes.



Height $\mathbf{3}$ full binary tree.

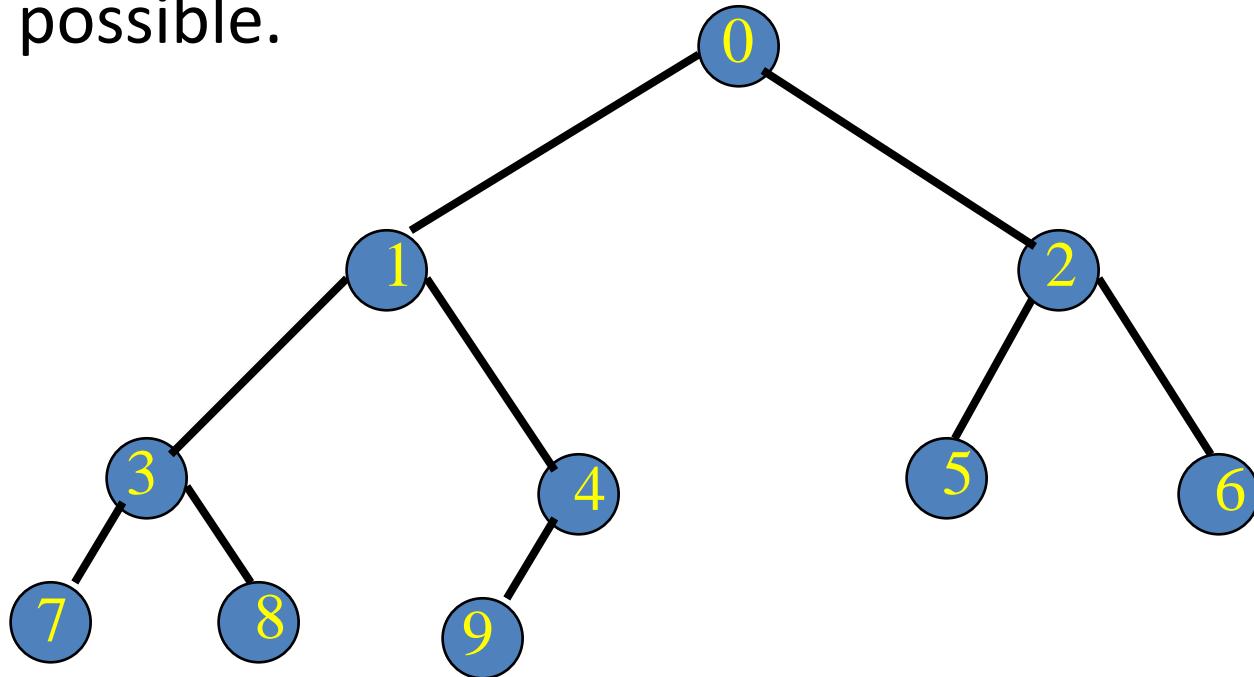
Numbering Nodes in a Full Binary Tree

- Number the nodes **0** through $2^{h+1} - 2$.
(total $2^{h+1} - 1$ nodes)
- Number by levels from top to bottom.
- Within a level number from left to right.



Complete Binary Tree

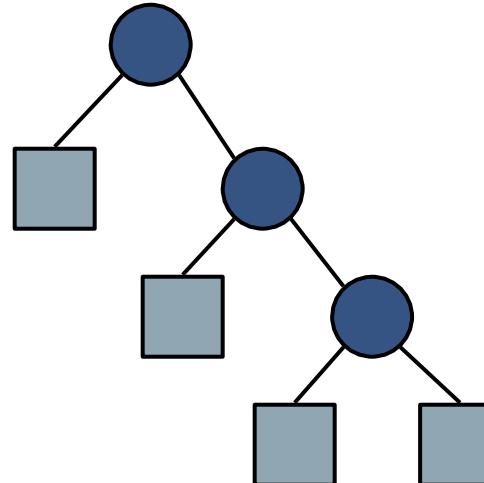
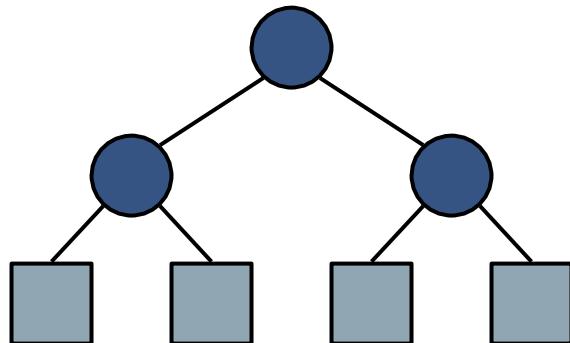
- all levels are completely filled except possibly the last level and the last level has all keys as **left** as possible.



- Example, Complete binary tree with 10 nodes.
- The size n of a complete binary tree of height h satisfies $2^h \leq n \leq 2^{h+1} - 1$

Proper Binary Trees

Meaning: Each internal node has exactly 2 children



Proper Binary Trees

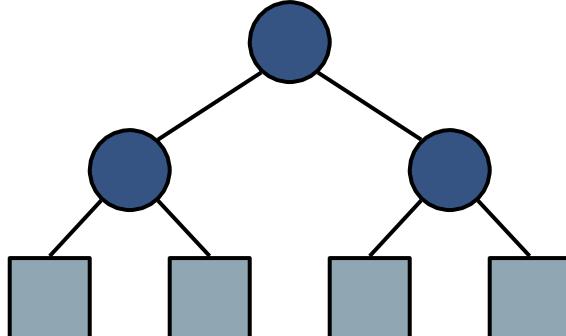
Given the following definitions:

n :number of total nodes

e :number of external nodes

i :number of internal nodes

h :height (maximum depth of a node)



Properties:

$$1. \ e = i + 1$$

$$2. \ n = 2e - 1$$

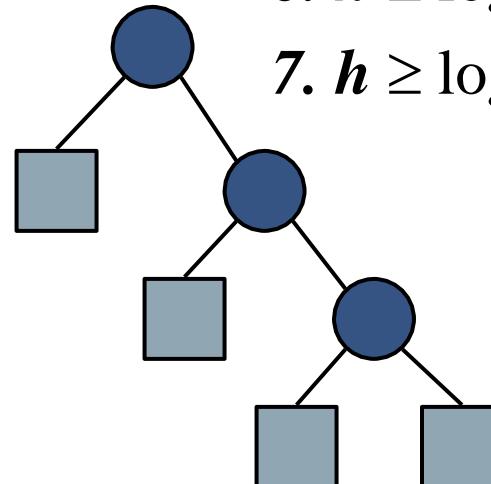
$$3. \ h \leq i$$

$$4. \ h \leq (n - 1)/2$$

$$5. \ e \leq 2^h$$

$$6. \ h \geq \log_2 e$$

$$7. \ h \geq \log_2 (n + 1) - 1$$

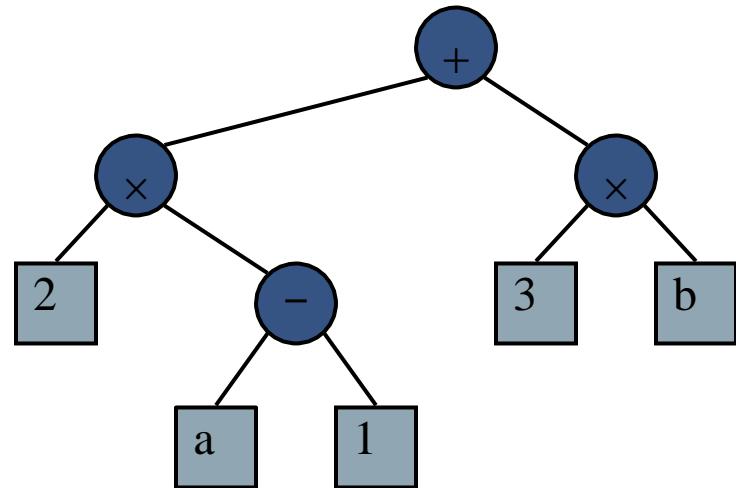


Examples of Usage of Binary Trees

Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
 - internal nodes: **operators**
 - external nodes: **operands**

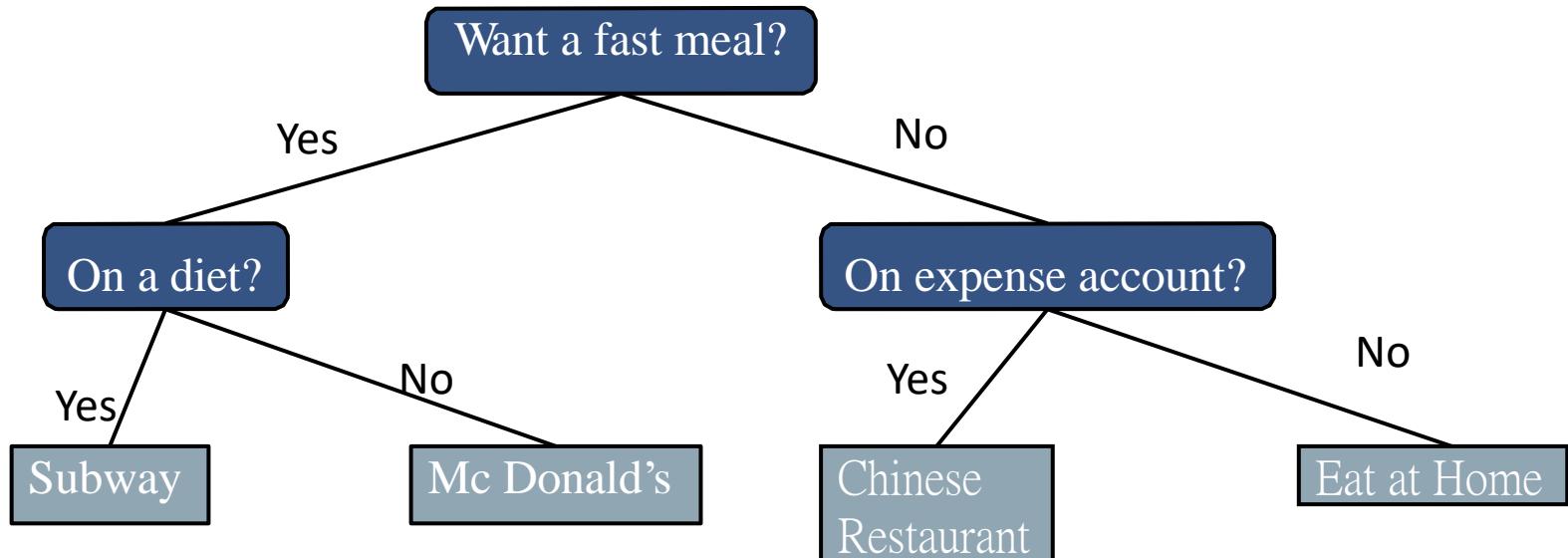
- Example: arithmetic expression tree for the expression:

$$(2 \times (a - 1) + (3 \times b))$$


Decision Tree

Binary tree associated with a decision process

- internal nodes: questions with yes/no answer
- external nodes: decisions
- Example: dining decision



Traversals of a Tree

Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree.
- In a traversal, each element of the binary tree is **visited** exactly once.
- During the **visit** of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

Binary Tree Traversal Methods

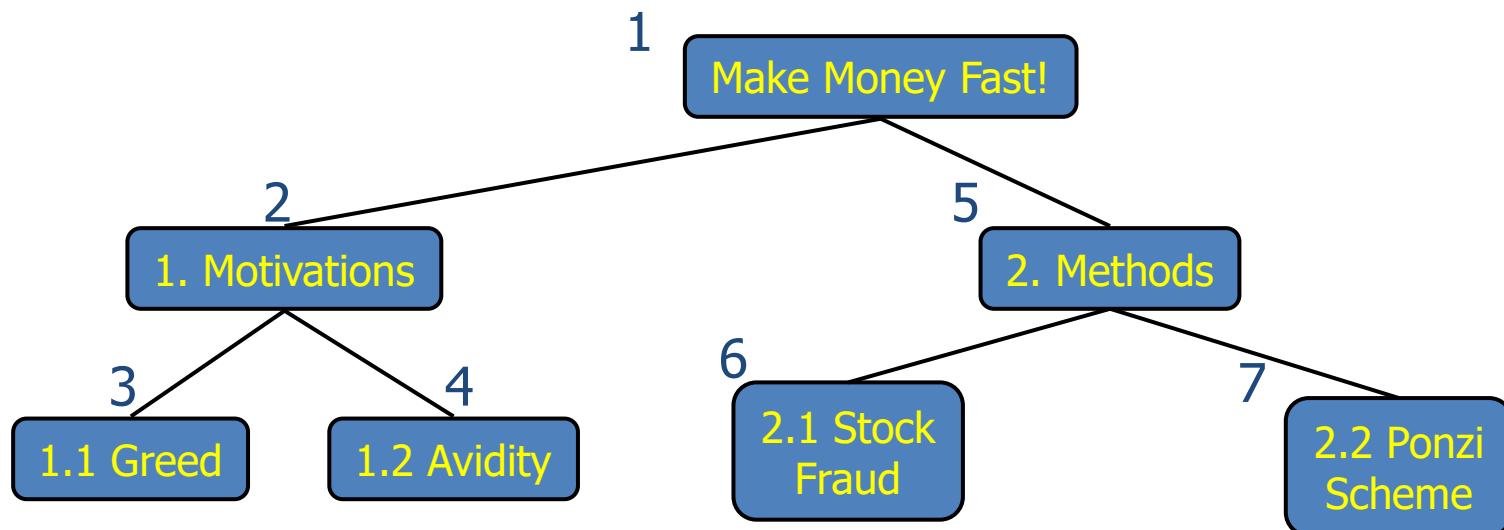
- Preorder (get prefix expression)
- Inorder (get infix expression)
- Postorder (get postfix expression)
- Level order

Preorder Traversals of a Binary Tree

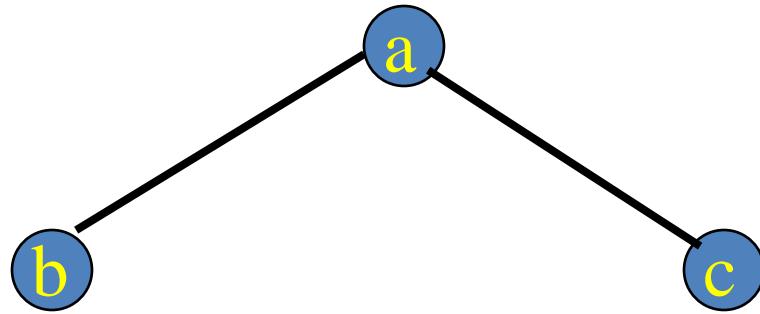
Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited **before its descendants**
- *Application:* print a structured document

```
Algorithm preOrder( $T, v$ )
    visit( $v$ )
    for each child  $w$  of  $v$ 
        preorder ( $T, w$ )
```

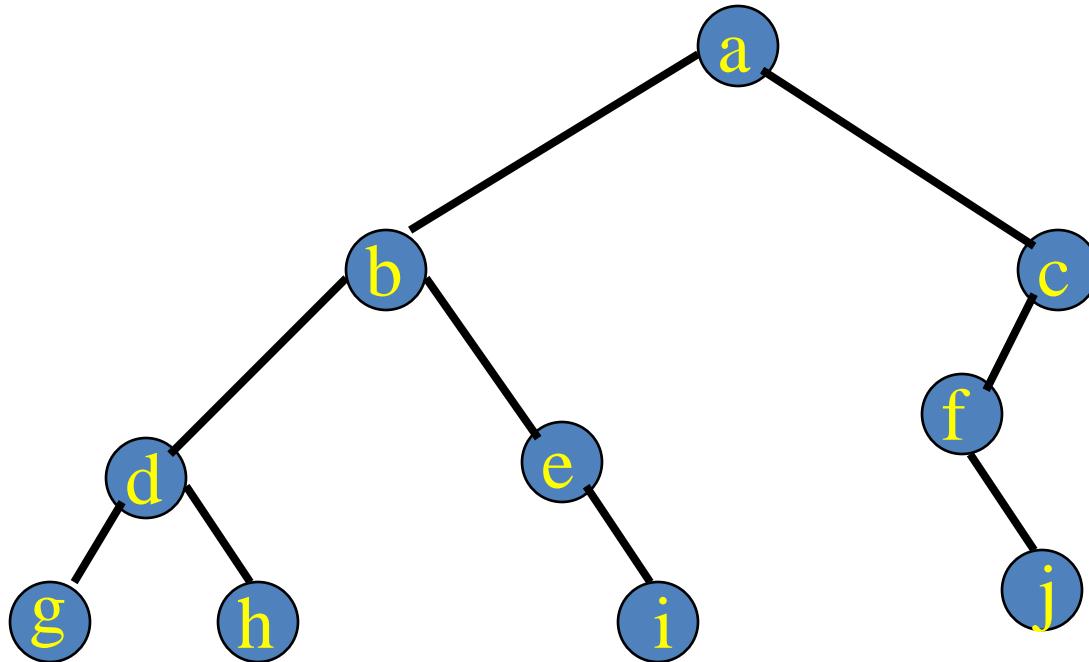


Preorder Example (visit = print)



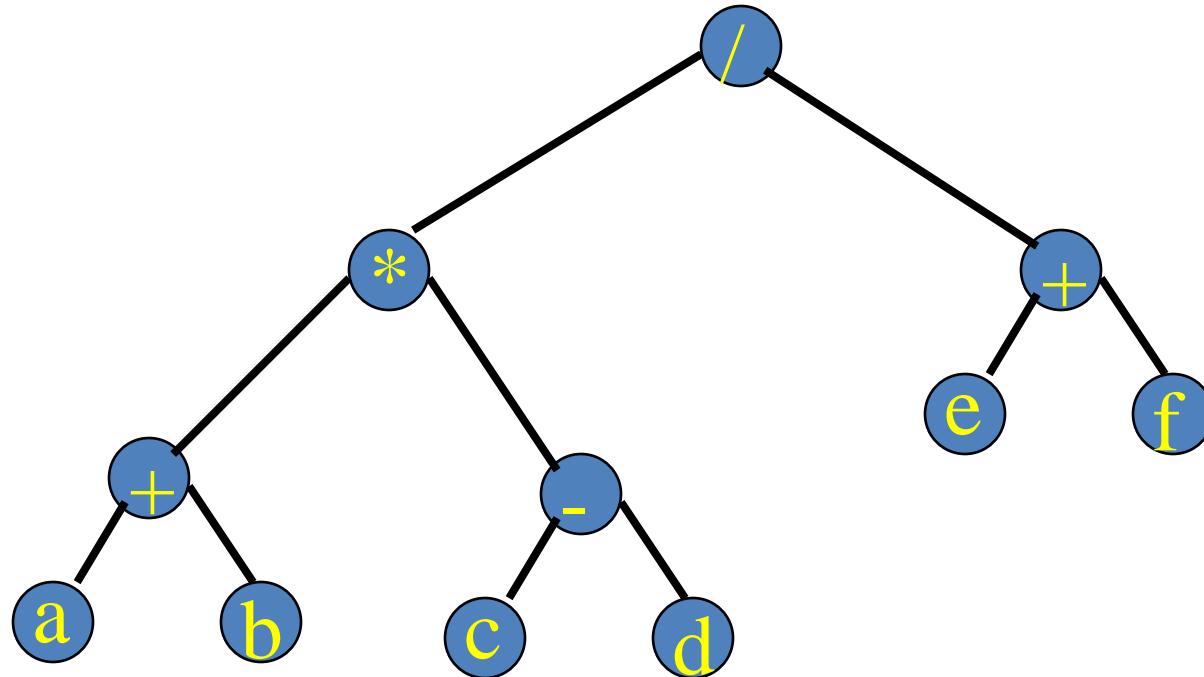
a b c

Preorder Example (visit = print)



a b d g h e i c f j

Preorder Of Expression Tree



/ * + a b - c d + e f

Gives **prefix** form of expression!

Preorder Traversal

```
def traversePreorder(self, root):  
    if root is not None:  
        print(root.data)  
        self.traversePreorder(root.left)  
        self.traversePreorder(root.right)
```

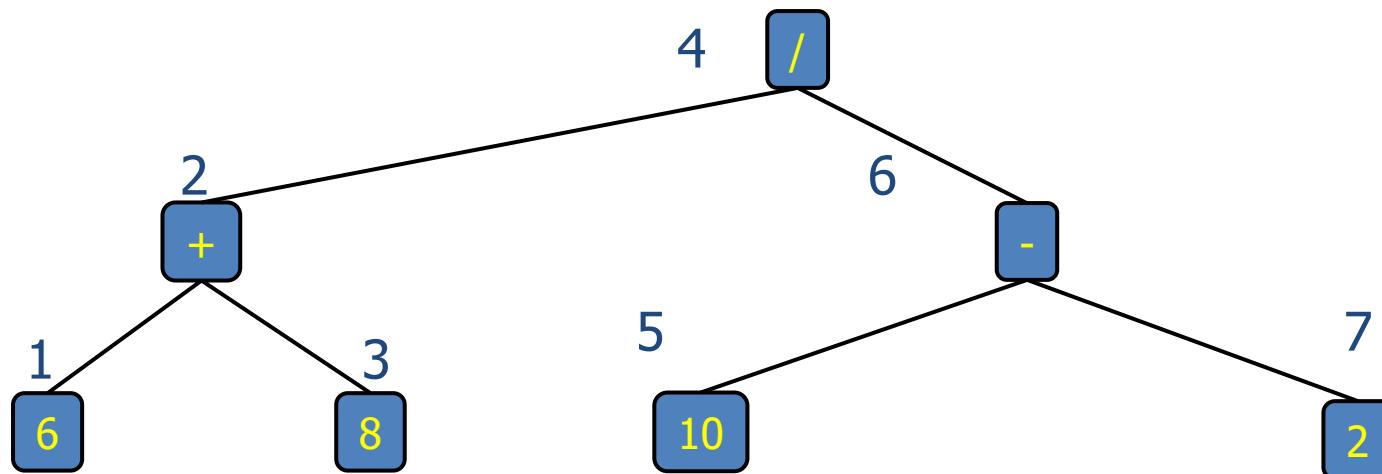
demo: BinaryTree2.py

Inorder Traversals of a Binary Tree

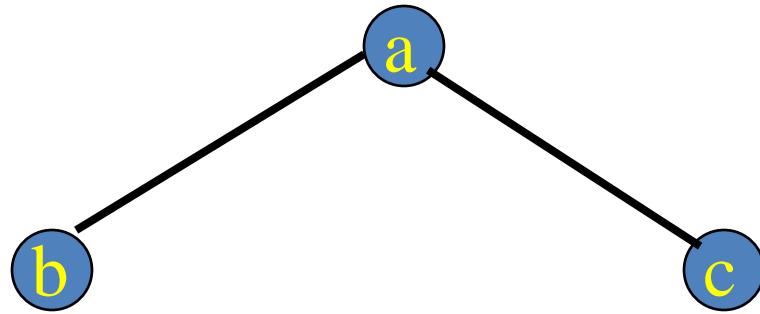
Inorder Traversal

- In an inorder traversal a node is visited **after its left subtree and before its right subtree**

```
Algorithm inOrder(v)
  if isInternal (v)
    inOrder (leftChild (v))
    visit(v)
    if isInternal (v)
      inOrder (rightChild (v))
```

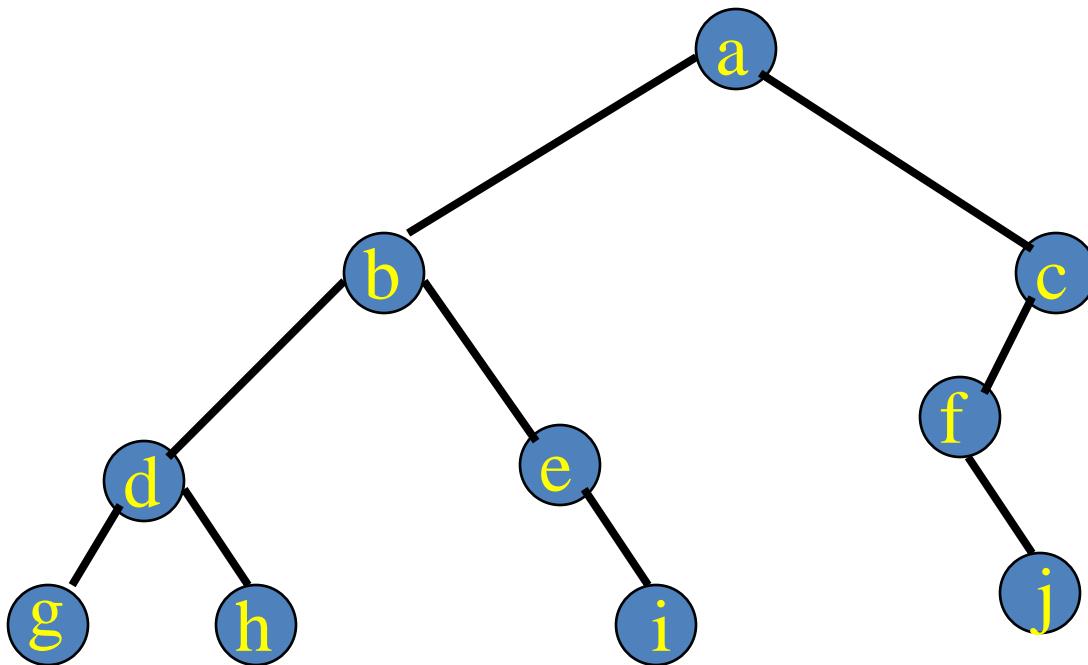


Inorder Example (visit = print)



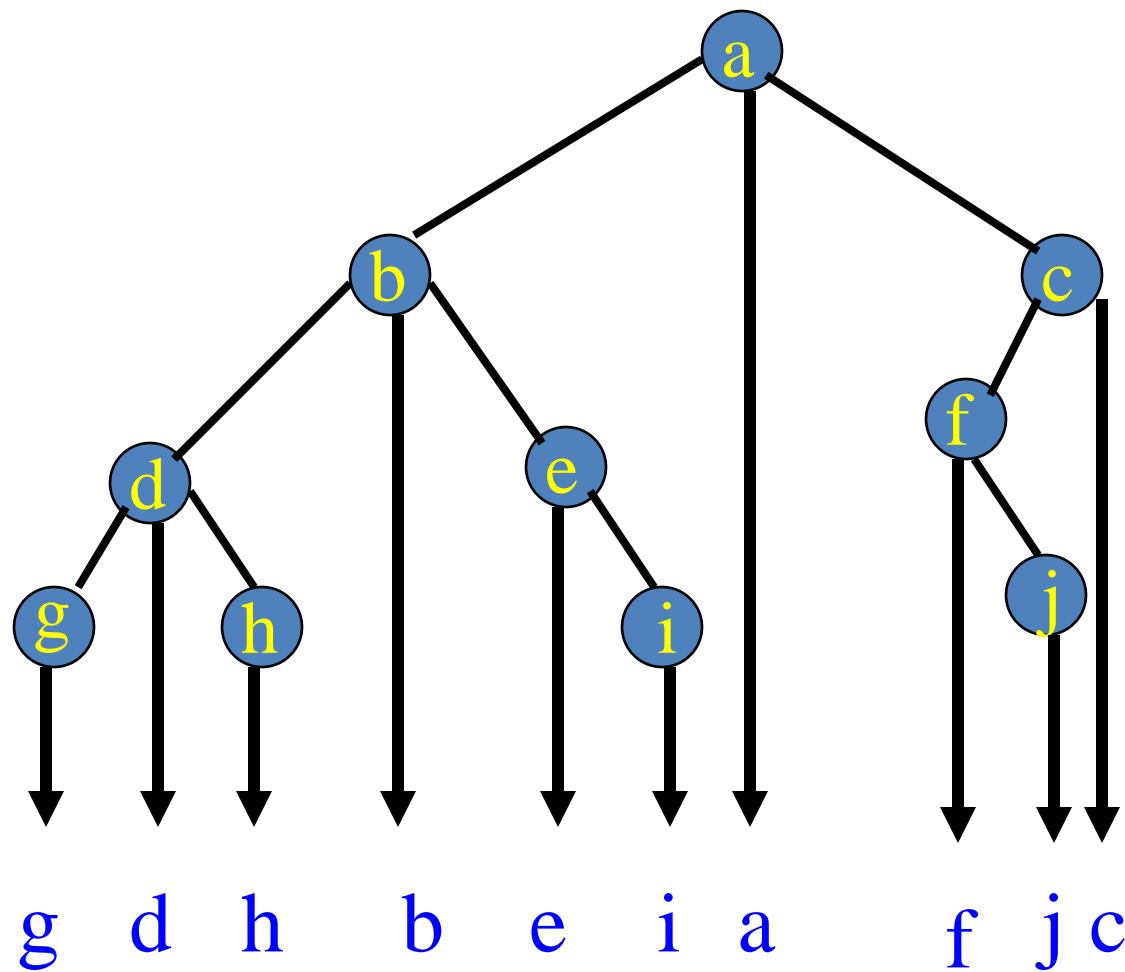
b a c

Inorder Example (visit = print)

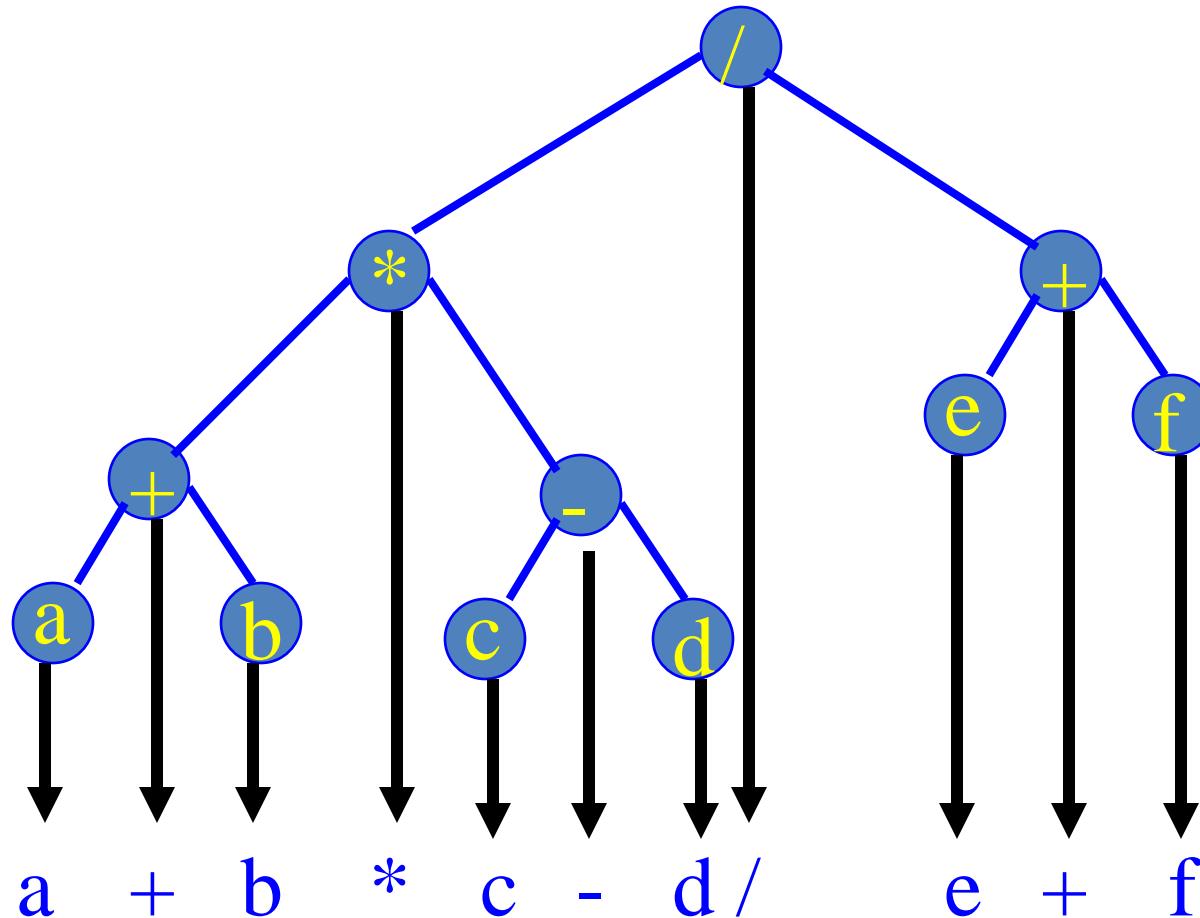


g d h b e i a f j c

Inorder By Projection



Inorder Of Expression Tree



Gives **infix** form of expression!

Inorder Traversal

```
def traverseInorder(self, root):  
    if root is not None:  
        self.traverseInorder(root.left)  
        print(root.data)  
        self.traverseInorder(root.right)
```

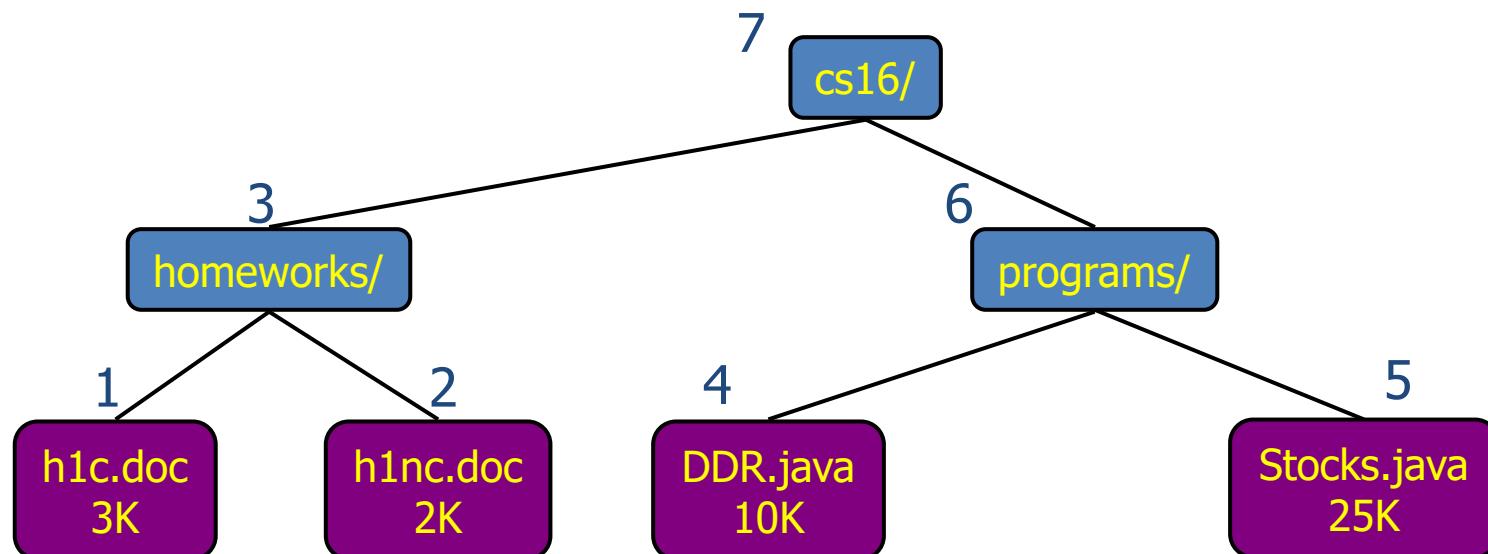
demo: BinaryTree2.py

Postorder Traversals of a Binary Tree

Postorder Traversal

- In a postorder traversal, a node is visited **after its descendants**
- Application: compute space used by files in a directory and its subdirectories

```
Algorithm postOrder(T,v)
for each child w of v
    postOrder (T,w)
    visit(v)
```

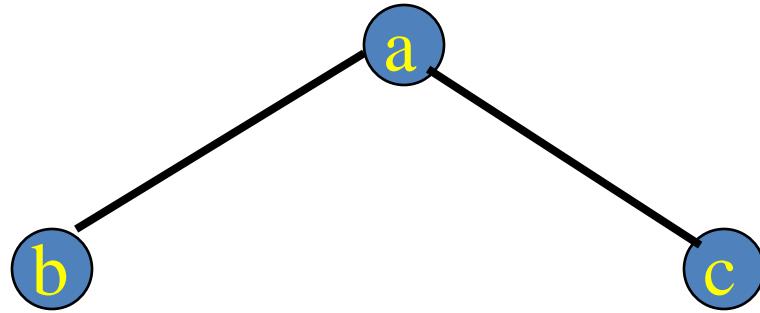


Postorder Traversal

```
def traversePostorder(self, root):  
    if root is not None:  
        self.traversePostorder(root.left)  
        self.traversePostorder(root.right)  
        print(root.data)
```

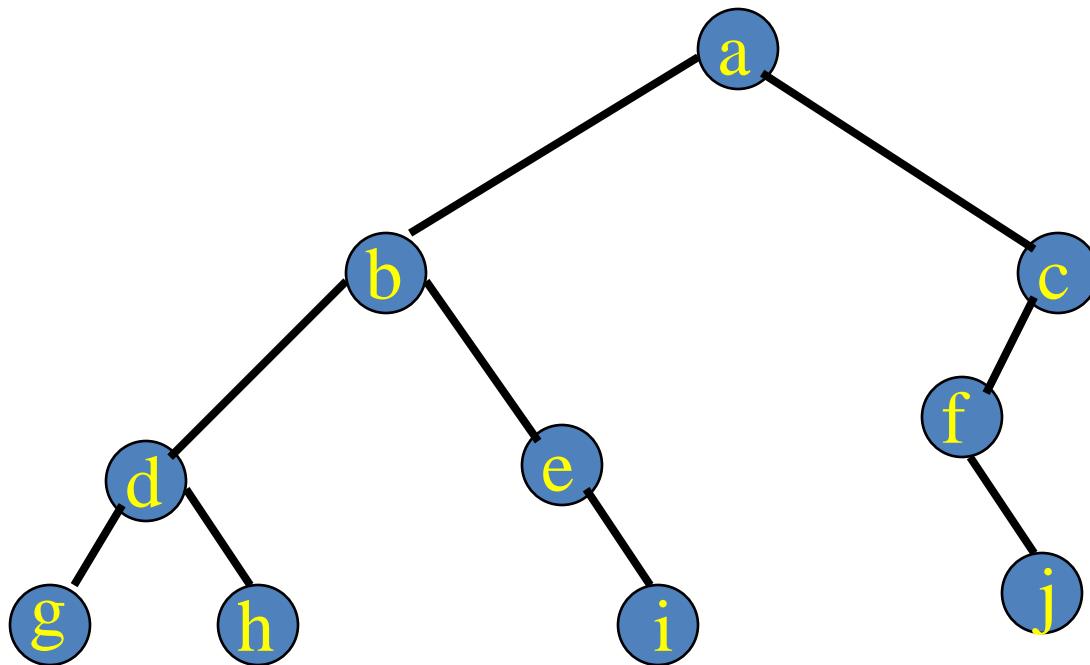
demo: BinaryTree2.py

Postorder Example (visit = print)



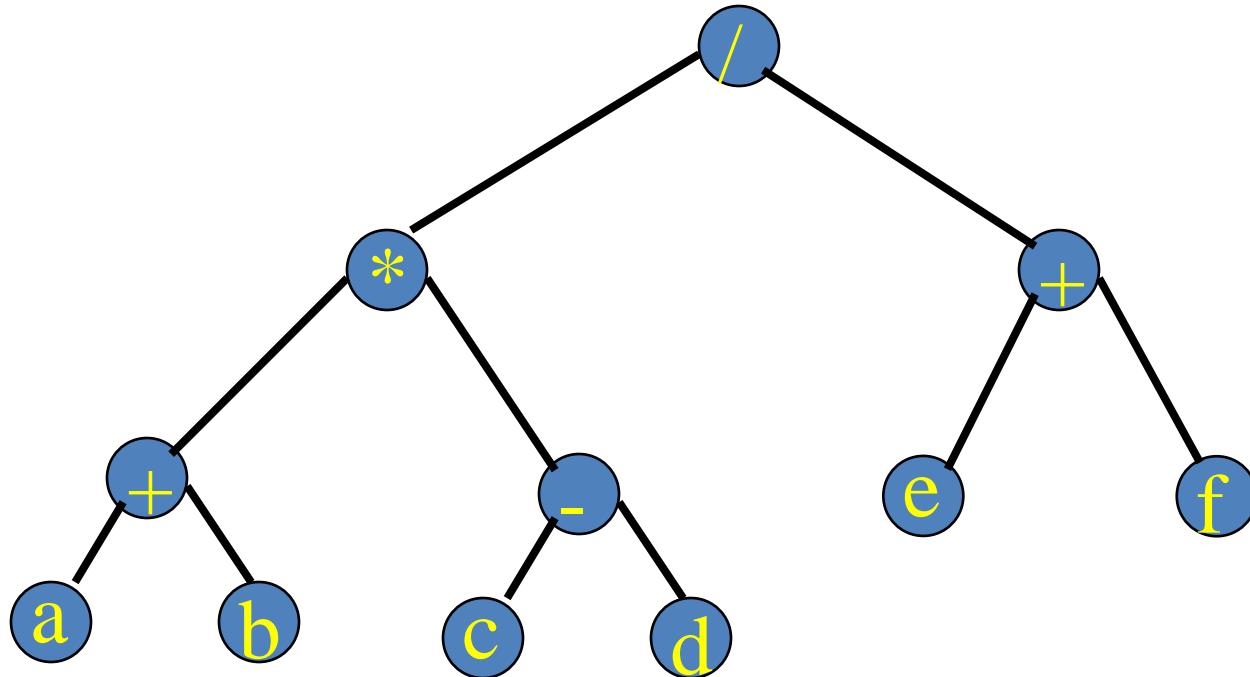
b c a

Postorder Example (visit = print)



g h d i e b j f c a

Postorder Of Expression Tree



a b + c d - * e f + /

Gives **postfix** form of expression!

Level order Traversals of a Binary Tree

Level Order

```
# Function to print level order traversal of tree
def printLevelOrder(self, root):
    h = root.height(root)
    for i in range(1, h+1):
        self.printCurrentLevel(root, i)

# Print nodes at a current level
def printCurrentLevel(self, root , level):
    if root is None:
        return
    if level == 1:
        print(root.data,end=" ")
    elif level > 1 :
        self.printCurrentLevel(root.left , level-1)
        self.printCurrentLevel(root.right , level-1)

# Compute the height of a tree--the number of nodes
# along the longest path from the root node down to
# the farthest leaf node

def height(self, node):
    if node is None:
        return 0
    else :
        # Compute the height of each subtree
        lheight = self.height(node.left)
        rheight = self.height(node.right)

        #Use the larger one
        if lheight > rheight :
            return lheight+1
        else:
            return rheight+1
```

demo: BinaryTree2

Level Order

levelorder(root)

 q = empty queue

 t = root

while not q.empty **do**

 visit(t)

if t.left ≠ null

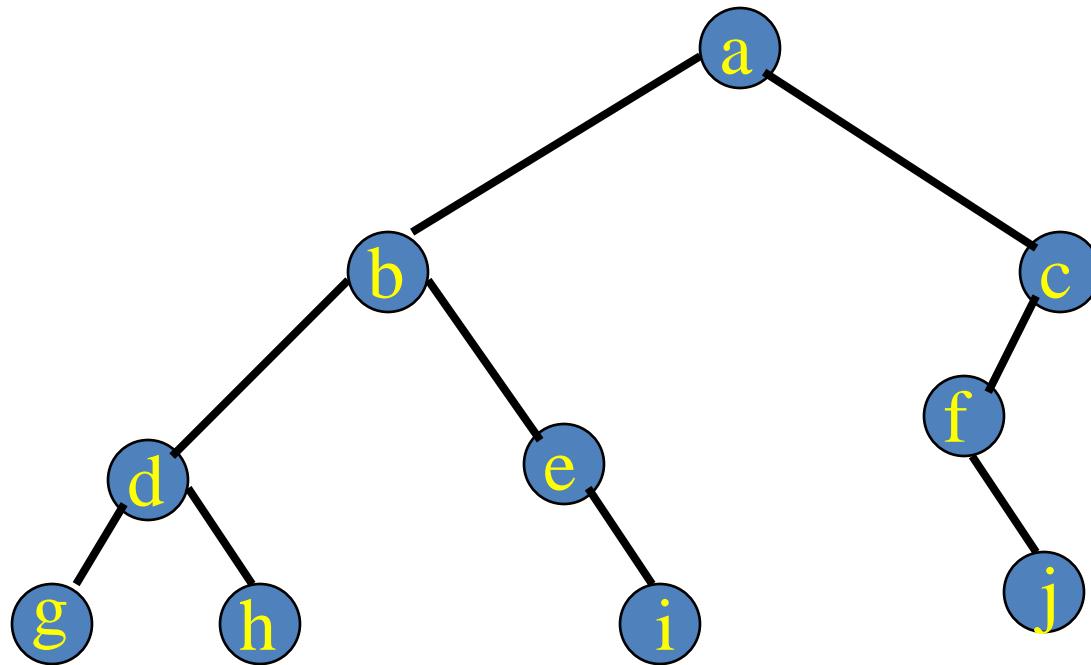
 q.enqueue(t.left)

if t.right ≠ null

 q.enqueue(t.right)

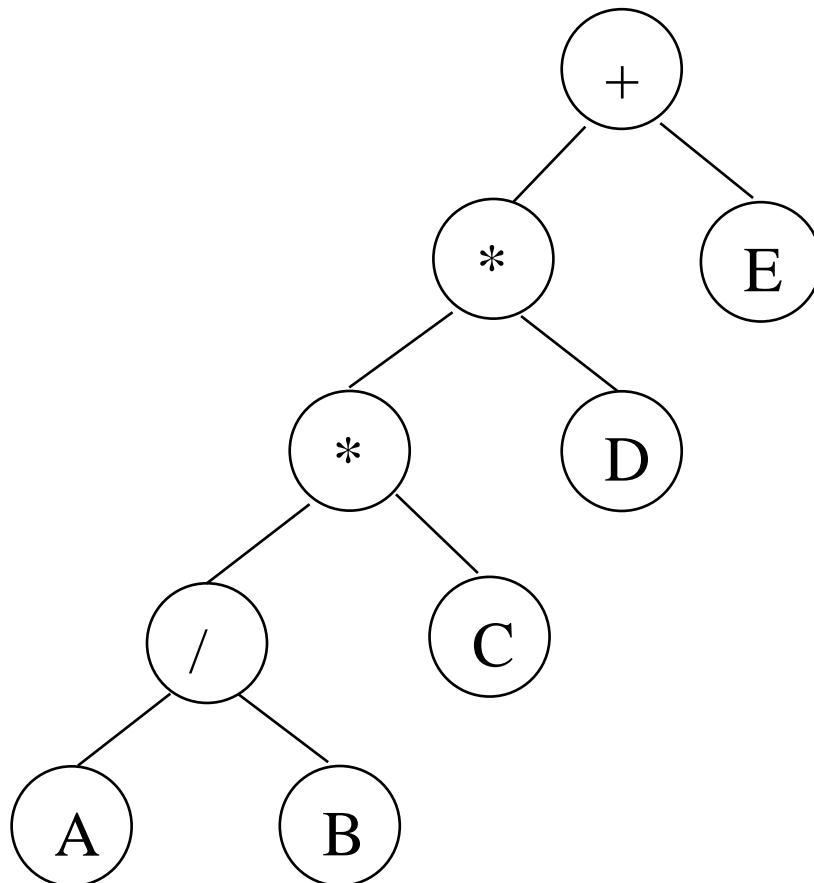
 t := q.dequeue()

Level-Order Example (visit = print)



a b c d e f g h i j

Arithmetic Expression Using Binary Tree



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

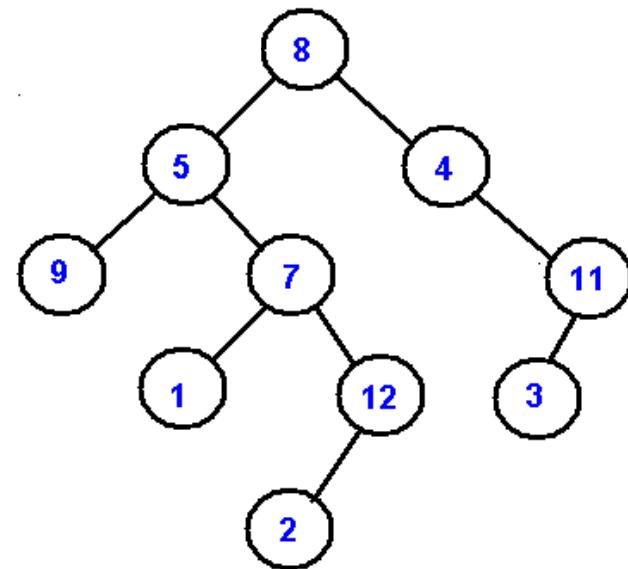
postfix expression

level order traversal

+ * E * D / C A B

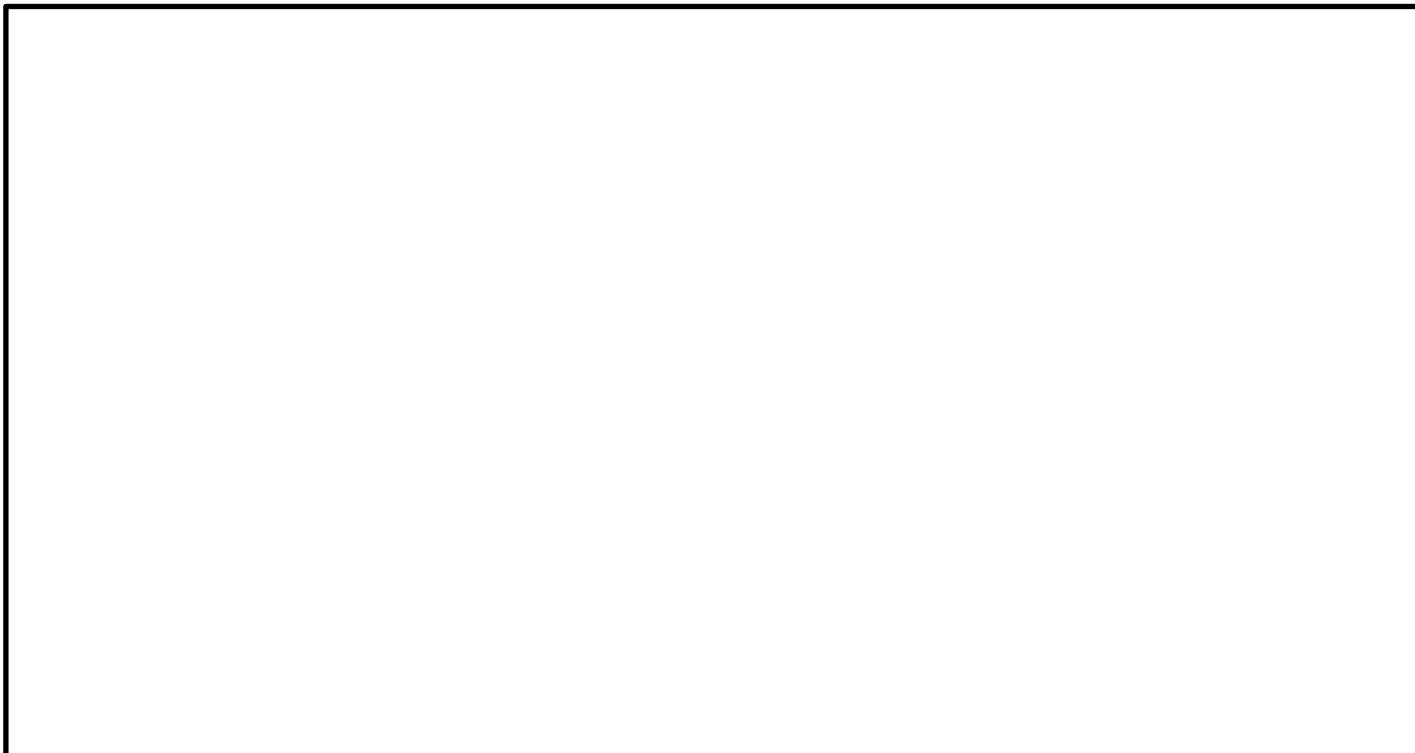
Exercise

- As an example consider the following tree and its four traversals:
- PreOrder –
- InOrder –
- PostOrder –
- LevelOrder –



Construct the tree

- Preorder: JCBADEFIGH
- In-order: ABCEDFJGIH



Summary

- General Trees
 - Creation of Trees
- Binary trees
 - Binary Tree Implementation
 - Full, Complete and proper
 - Application: e.g. Decision Trees
- Tree Traversal
 - Pre-order,
 - postorder,
 - inorder or
 - level order

SEHH2239
Data Structures

Lecture 9

Tree II



Learning Objectives:

- To describe the Binary Search Trees (BST)
- To implement BST by using a linked chain
- To use BST for searching and sorting
- To analysis BST and realize the need of balanced BST
- To operate the AVL Tree

Binary Search Tree

Key-value pair and Search Tree

- **Key-value pair (KVP) is a set of two data items:**
 - **Key, which is a unique identifier for some item of data, and**
 - **Value, which is either the *data that is identified* or a *pointer to the location of that data***
 - **Examples**
 - (0, Apple), (1, Orange), (2, Pineapple),....
 - (002, “Chan Tia”), (004, “U Pei Yi”), (007, “Lee Chi Gi”)....
 - (“atmosphere”, “環境”), (“believe”, “相信”) (“mood”, 心境 · 心情 · 情緒 ; 精神狀態”), (“process”, “過程 ; 步驟”)
- **A Search Tree is a tree data structures that can be used to perform searching from a (key, value) pair.**

Definition Of Binary Search Tree

- A binary tree.
- Each node has a **(key, value)** pair.
- Nodes are inserted to the tree according to the key.
- For every node **x**, all **keys** in the **left** subtree of **x** are **smaller** than the key in **x**.
- For every node **x**, all **keys** in the **right** subtree of **x** are **greater** than the key in **x**.
- **No duplicate nodes.**

Binary Search Trees

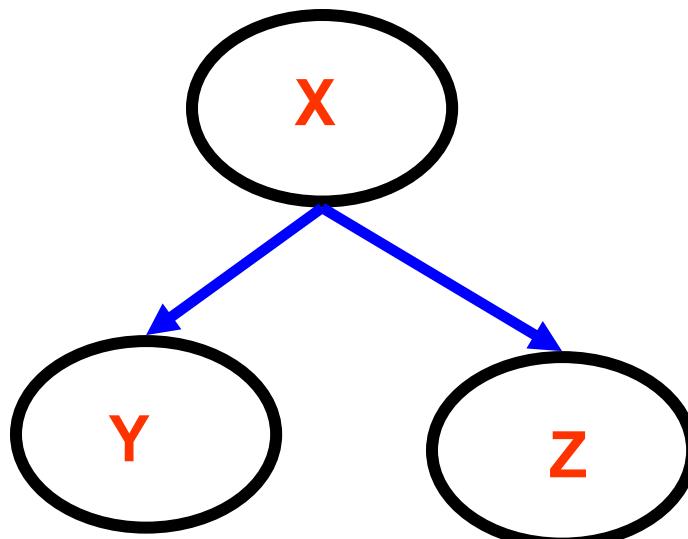
■ Key property

■ Value at node

- Smaller values in left subtree
- Larger values in right subtree

■ Example

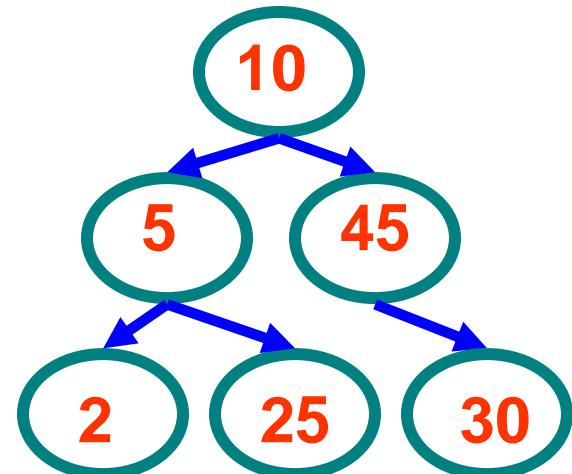
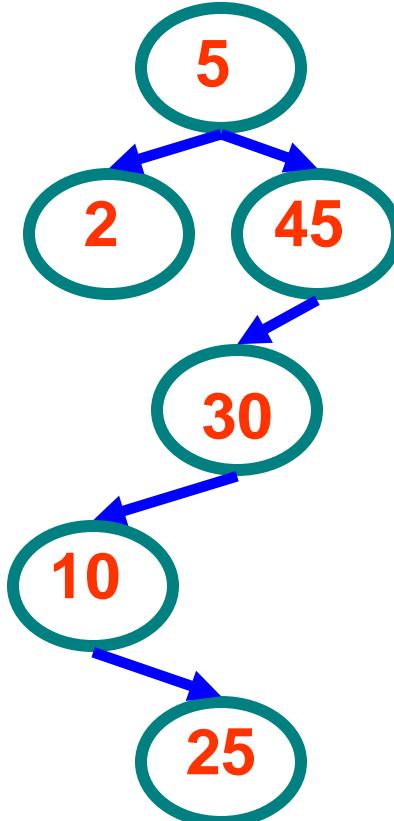
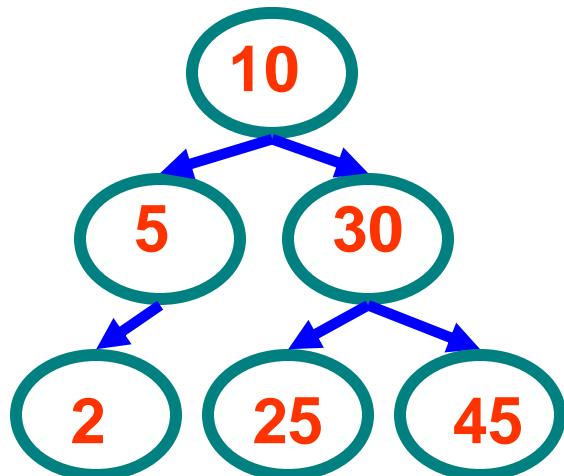
- $X > Y$
- $X < Z$



<https://www.youtube.com/watch?v=mtvbVLK5xDQ>
(00:00 – 02:00)

Binary Search Trees

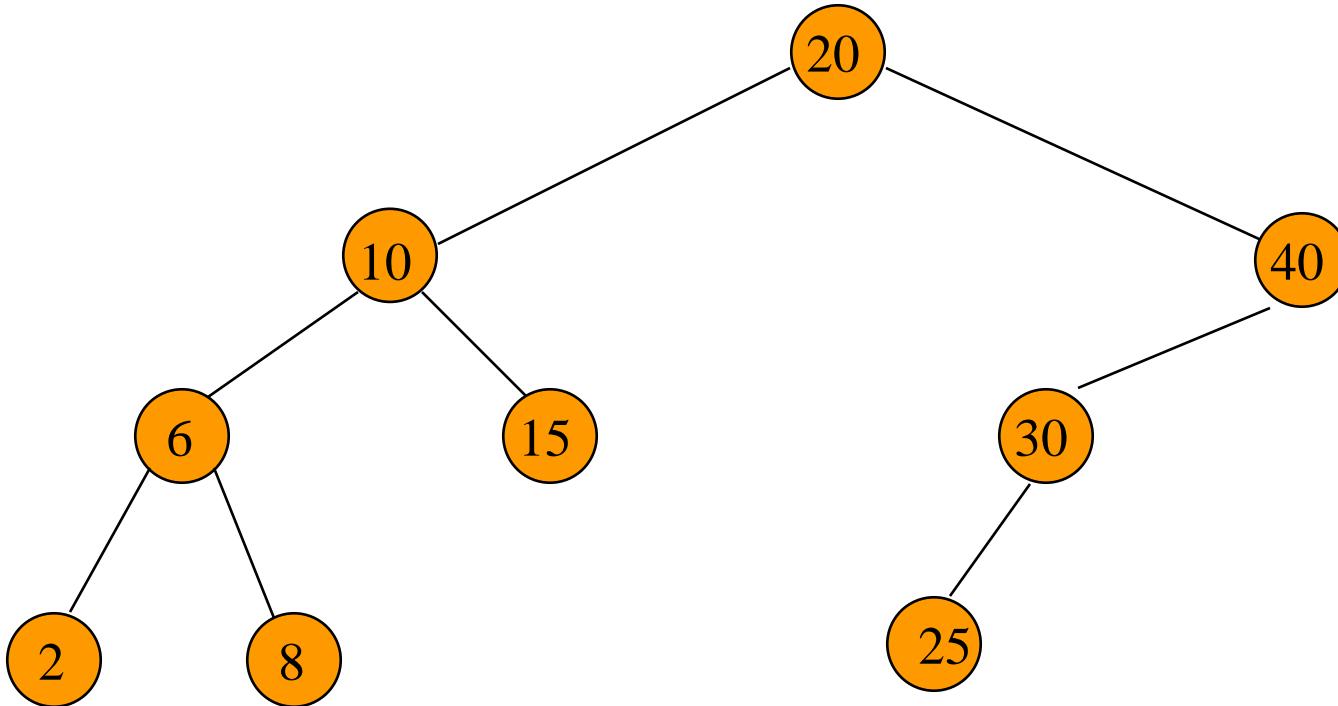
■ Examples



Binary
search trees

Non-binary
search tree

Example Binary Search Tree

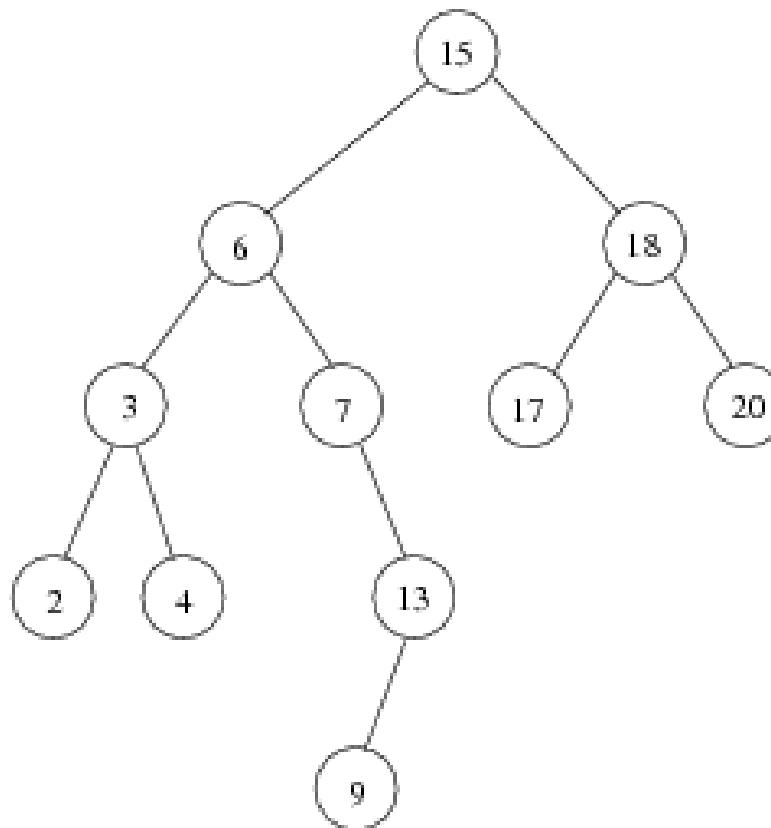


Insert 20, 10, 6, 2, 8, 15, 40, 30, 25

Only keys are shown.

Inorder traversal of BST

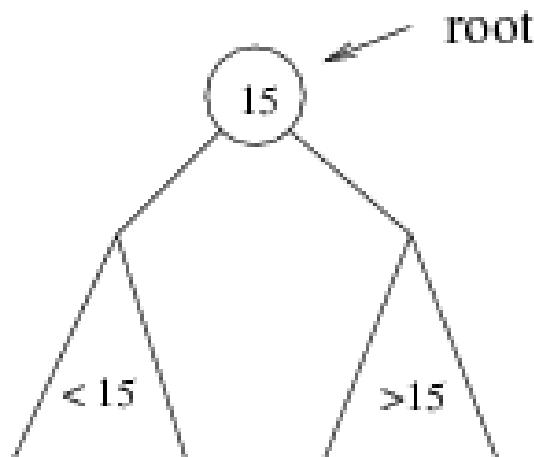
- Print out all the keys in sorted order



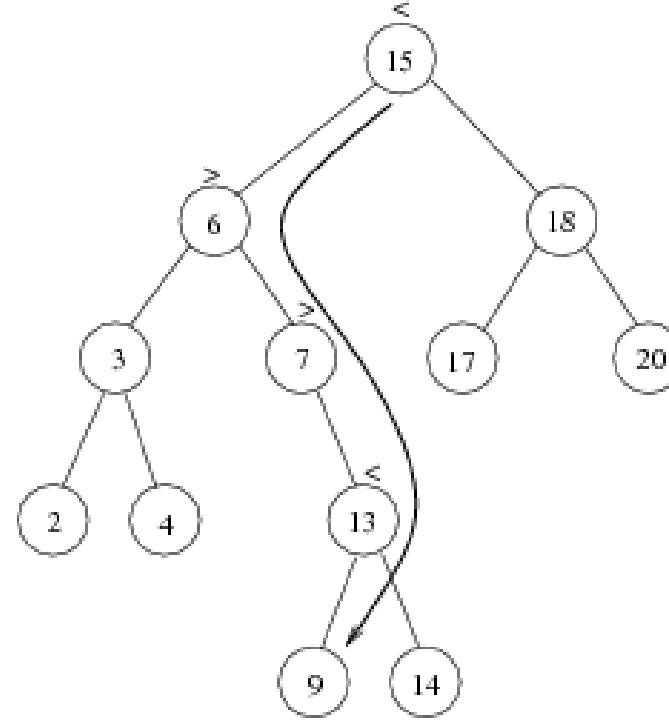
Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15 , then we should search in the left subtree.
- If we are searching for a key > 15 , then we should search in the right subtree.



Example: Search for 9 ...



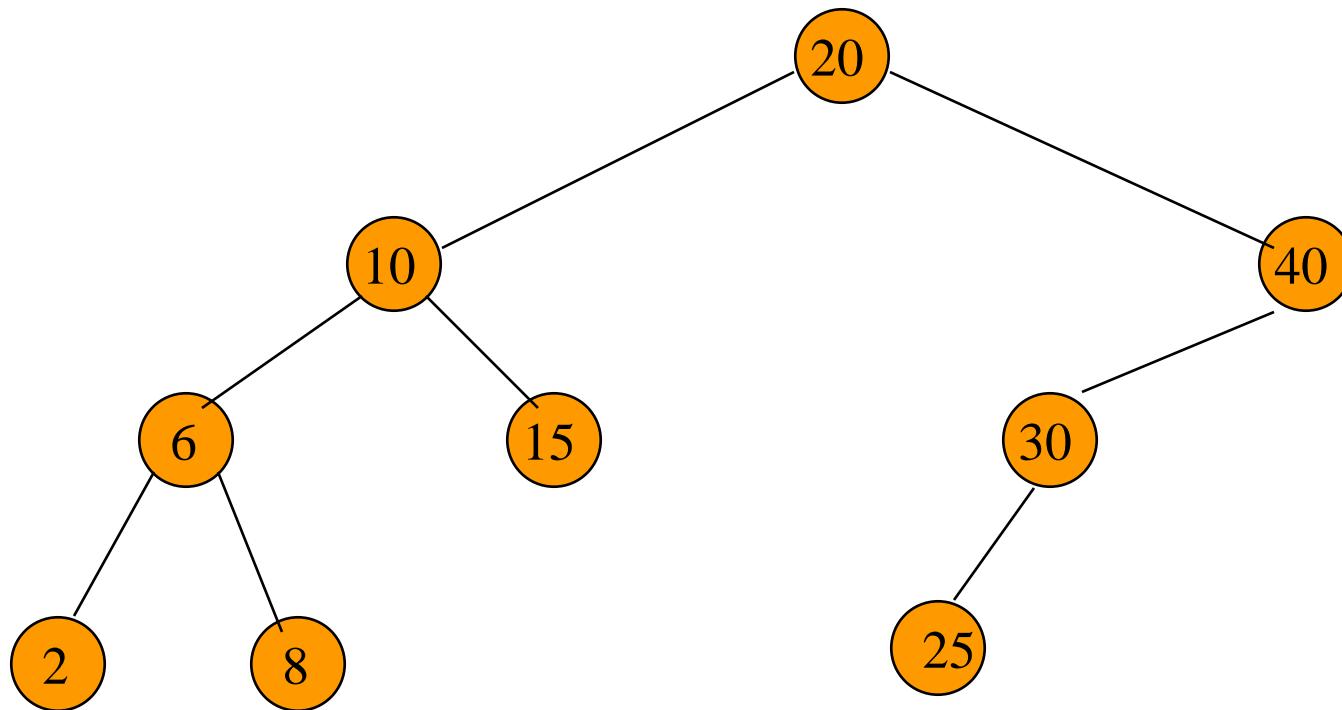
Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Key operations of BST

- $\text{get}(k)$
 - get the value v for a given key k
- $\text{put}(k, v)$
 - add a new (k, v) to a BST
- $\text{remove}(k)$
 - delete a node with key k from BST

The Operation get()

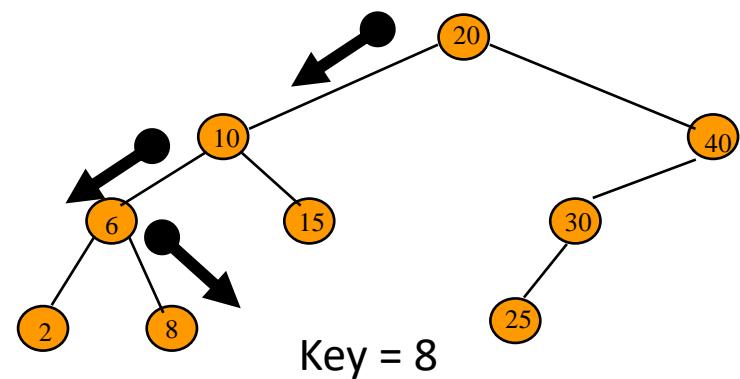
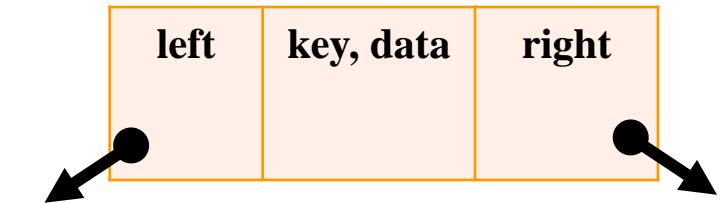


The Operation `get()`

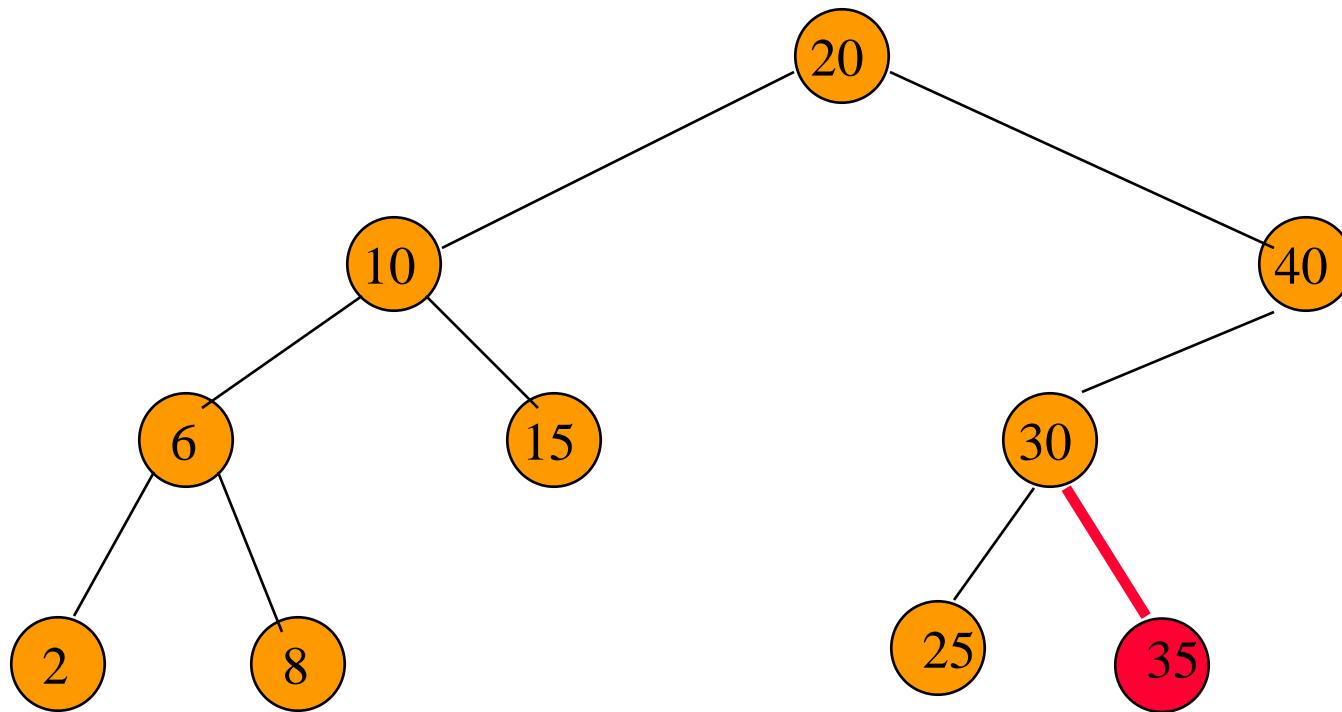
```
class Node:  
    def __init__(self, key, data):  
        self.left = None  
        self.right = None  
        self.key = key  
        self.data = data
```

```
def get(self, key):  
    p = self  
  
    while p is not None:  
        if p.key < key:  
            p = p.left  
        else:  
            if p.key > key:  
                p = p.right  
            else:  
                return p.data
```

```
#no matching key  
return None
```

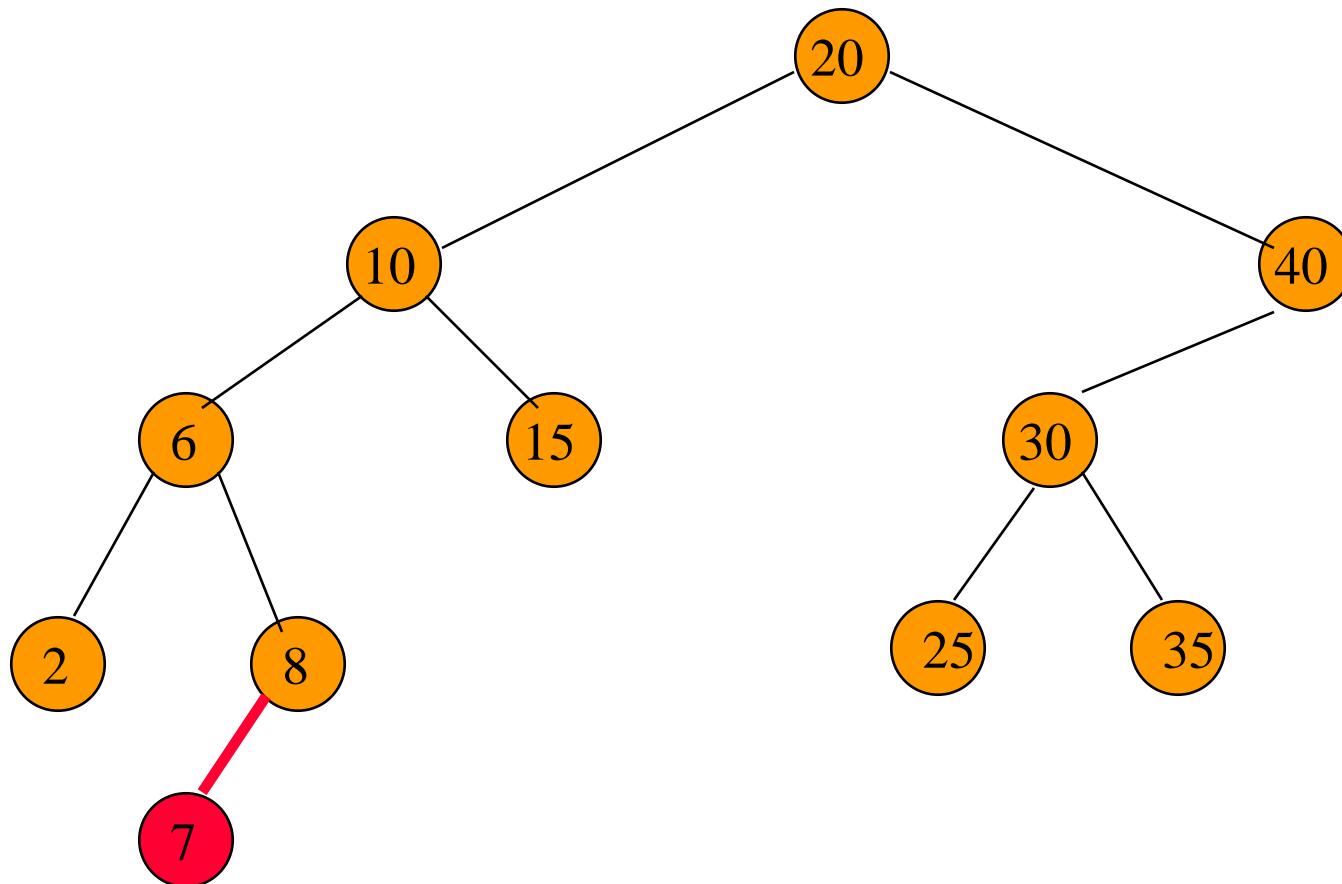


The Operation put()



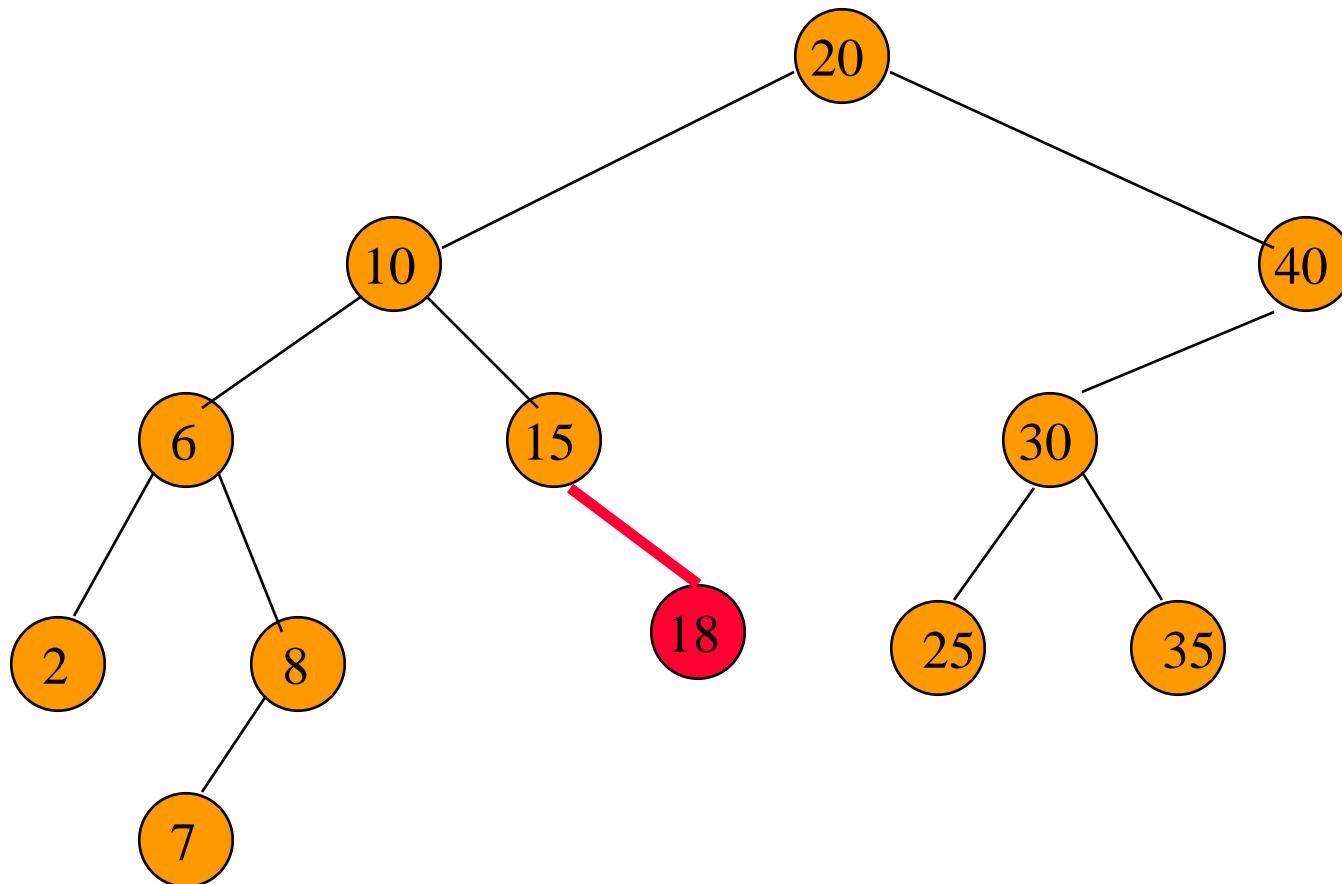
Put a pair whose key is 35.

The Operation put()



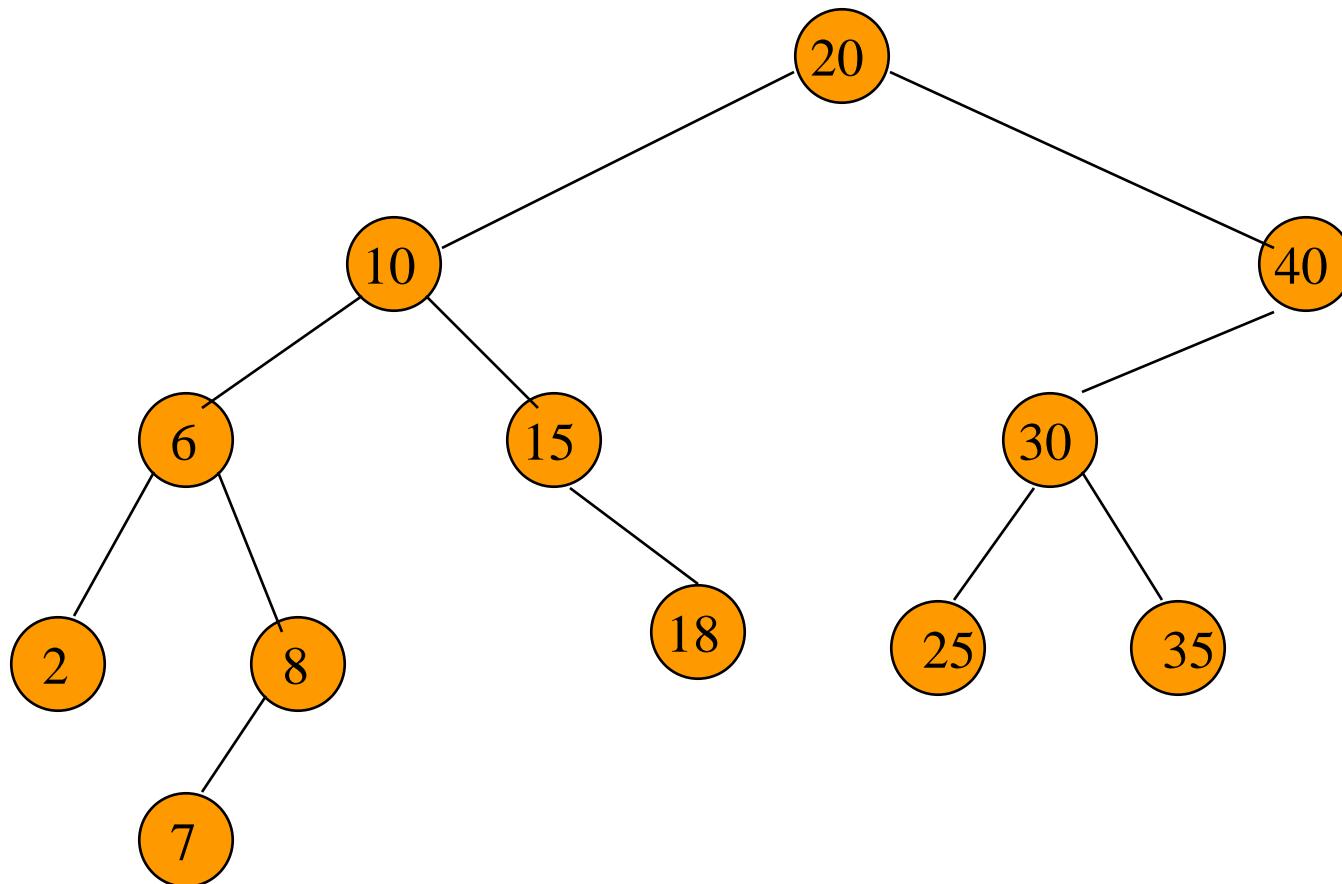
Put a pair whose key is 7.

The Operation put()



Put a pair whose key is 18.

The Operation put()



The Operation put()

```
def put(self, key, data):
    # Compare the new value with the parent node
    if self.key:
        if key < self.key:
            if self.left is None:
                self.left = Node(key, data)
            else:
                self.left.put(key, data)
        elif key > self.key:
            if self.right is None:
                self.right = Node(key, data)
            else:
                self.right.put(key, data)
        else:
            self.key = key
            self.data = data
```

Exercise

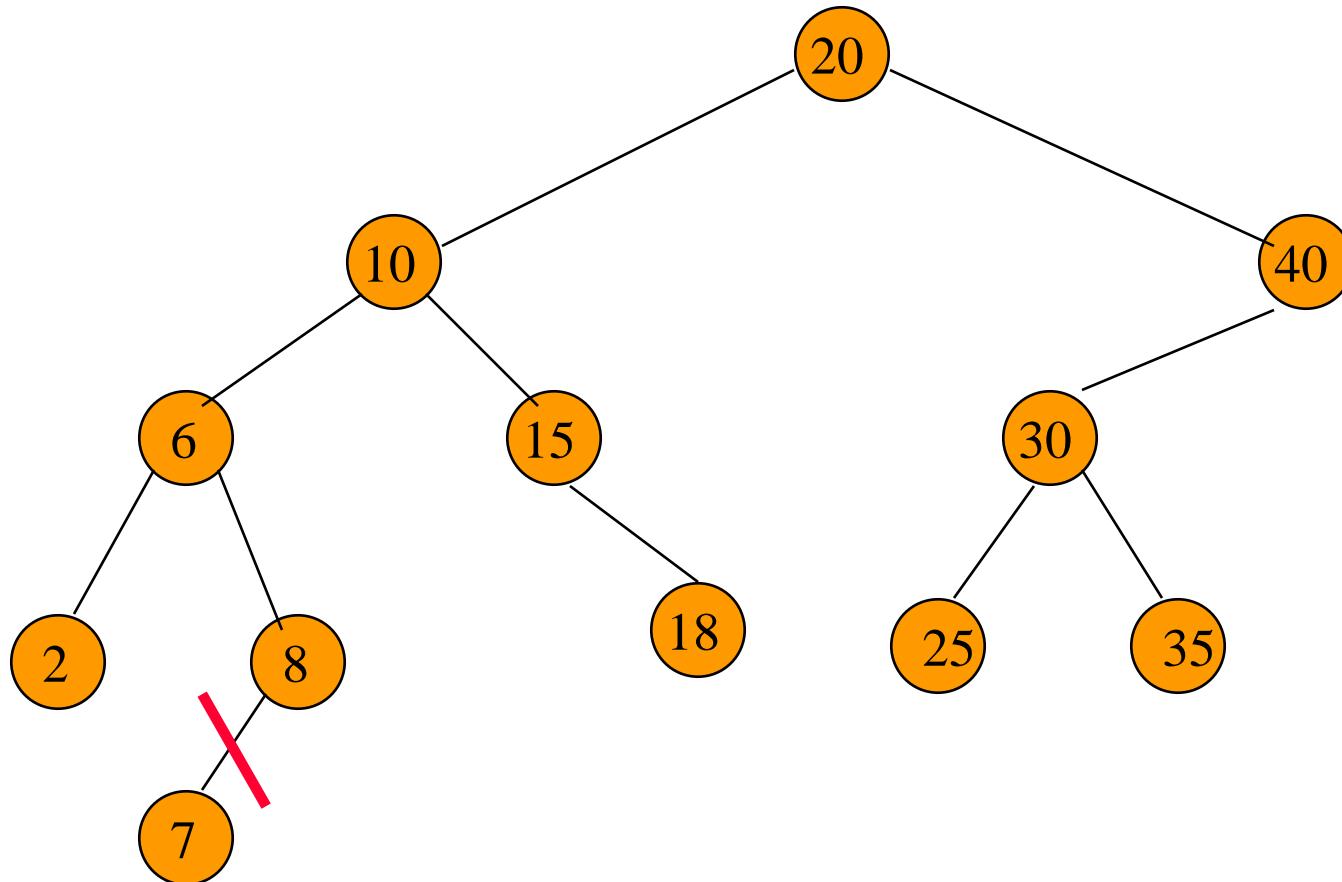
- For each of the following key sequences determine the binary search tree obtained when the keys are inserted one-by-one in the order given into an initially empty tree:
 - A. 1, 2, 3, 4, 5, 6, 7.
 - B. 4, 2, 1, 3, 6, 5, 7.
 - C. 1, 6, 7, 2, 4, 3, 5.

The Operation remove()

Three cases:

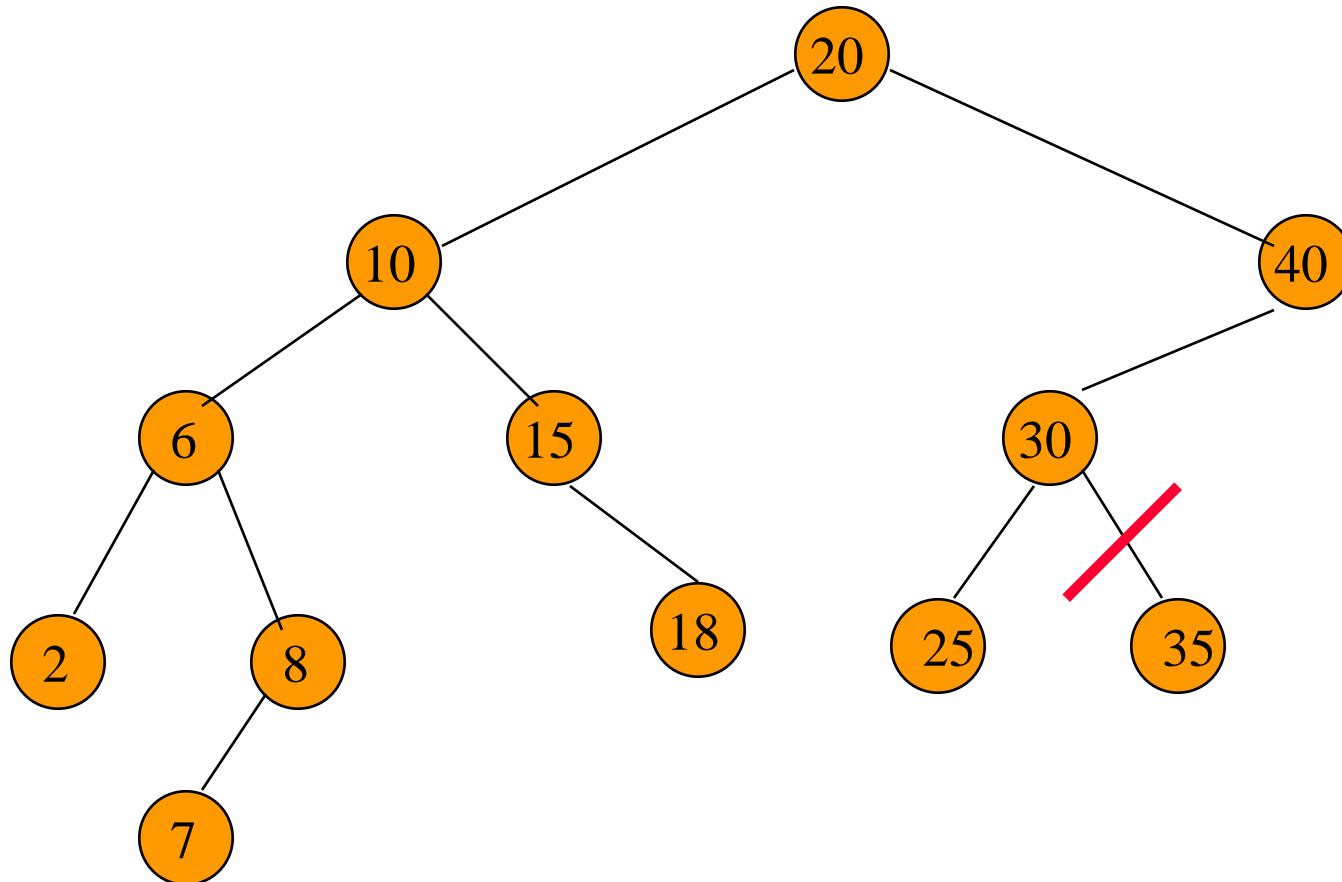
- Element is in a leaf.
- Element is in a degree 1 node.
- Element is in a degree 2 node.

Remove From A Leaf



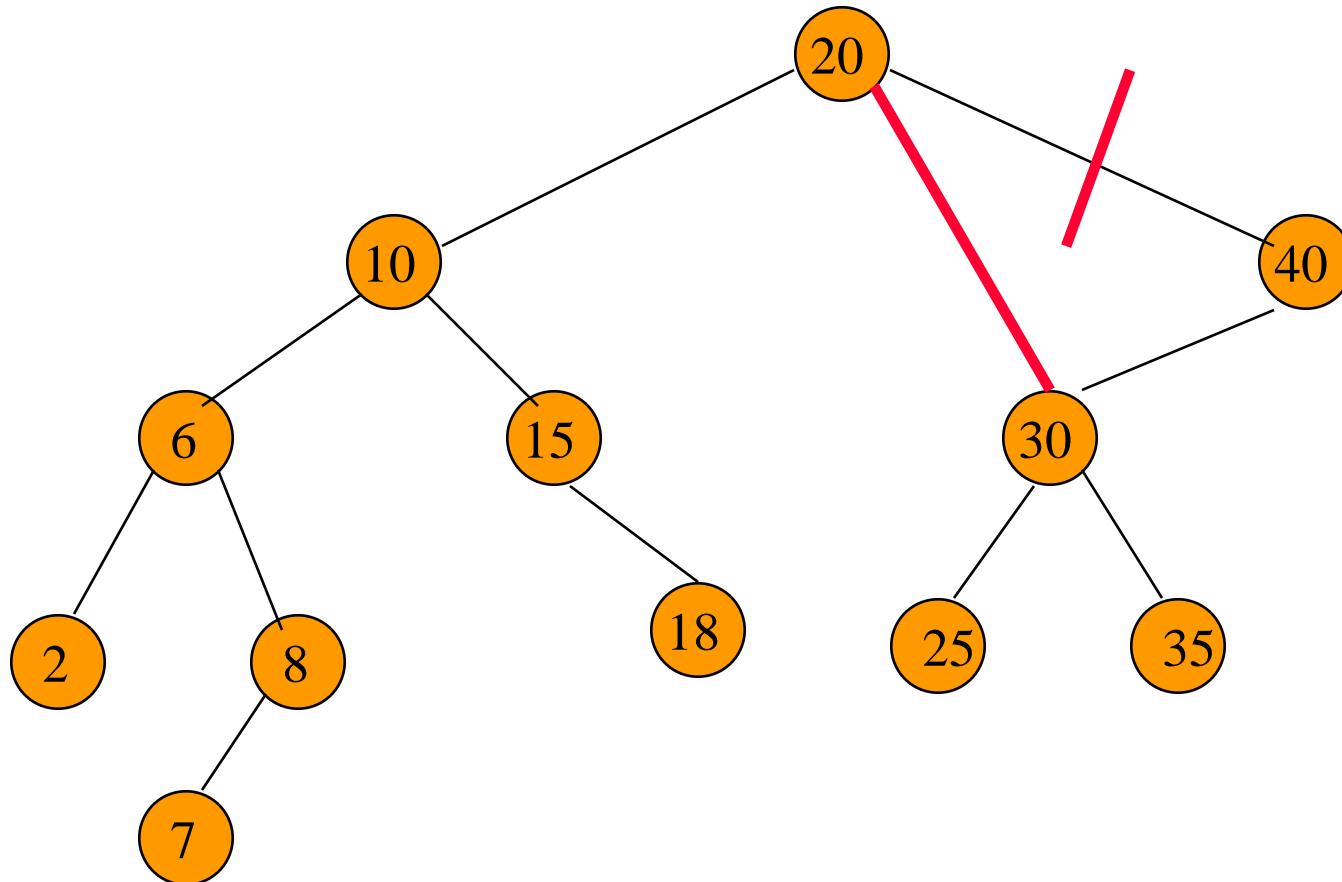
Remove a leaf element. key = 7

Remove From A Leaf (contd.)



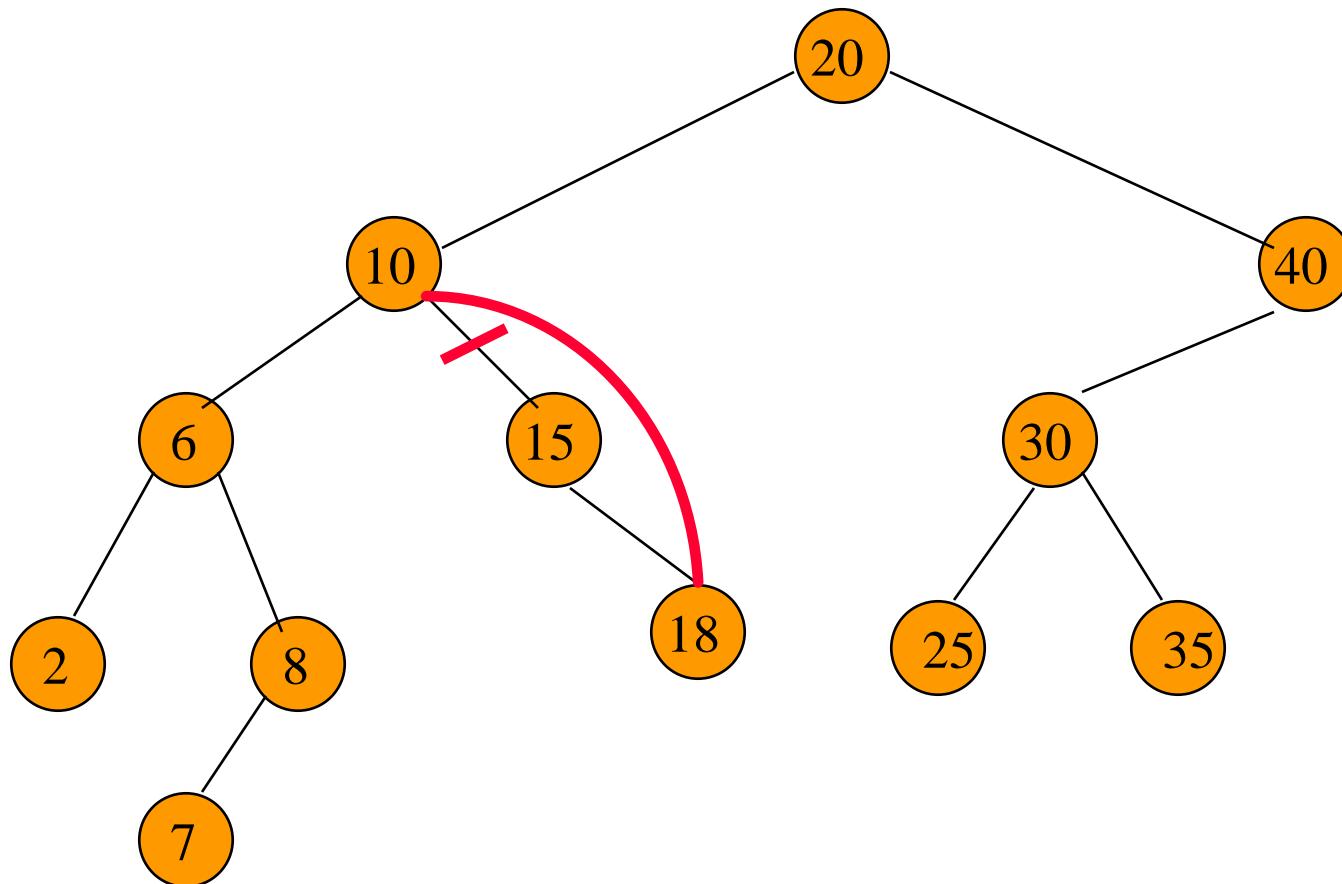
Remove a leaf element. key = 35

Remove From A Degree 1 Node



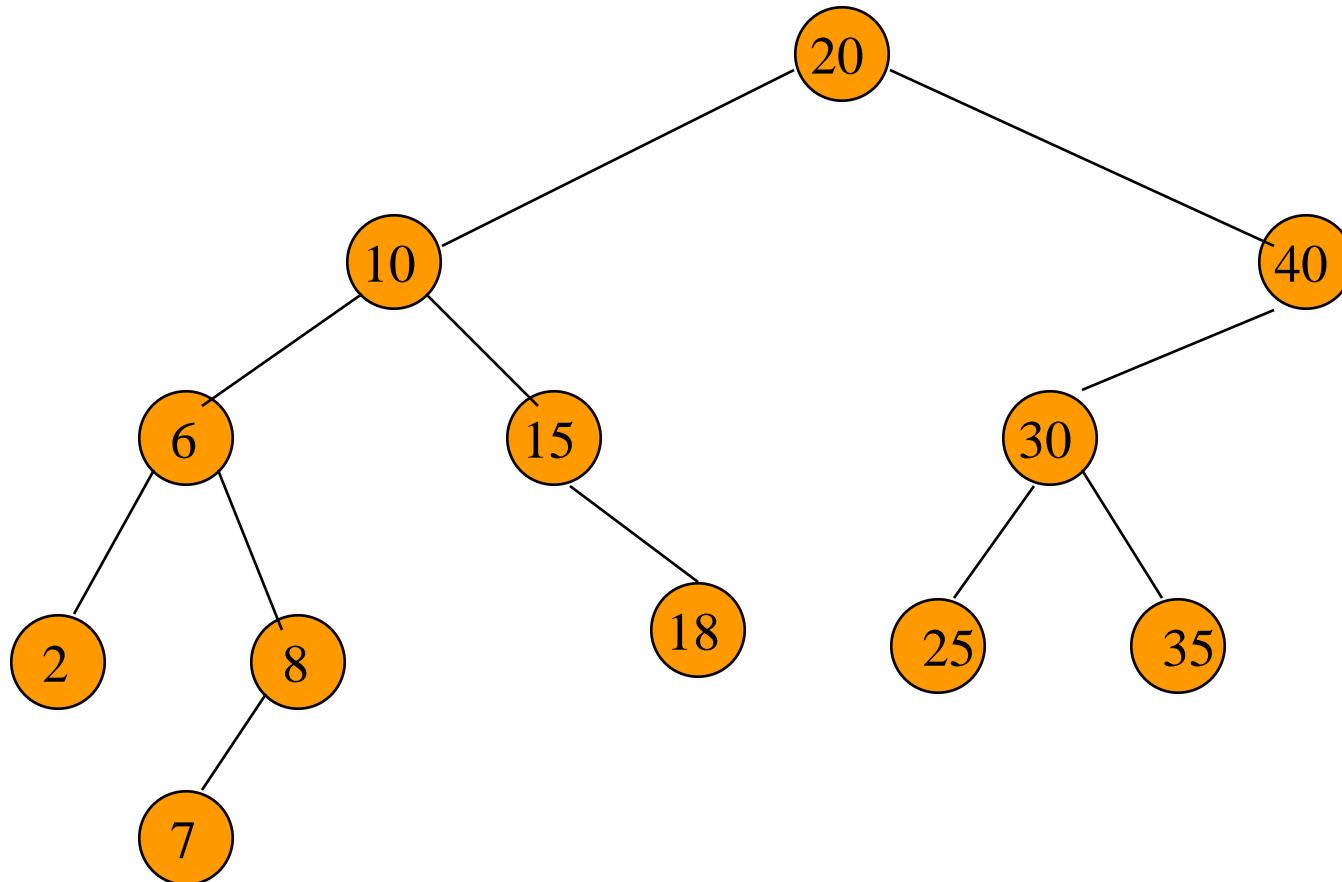
Remove from a degree 1 node. key = 40

Remove From A Degree 1 Node (contd.)



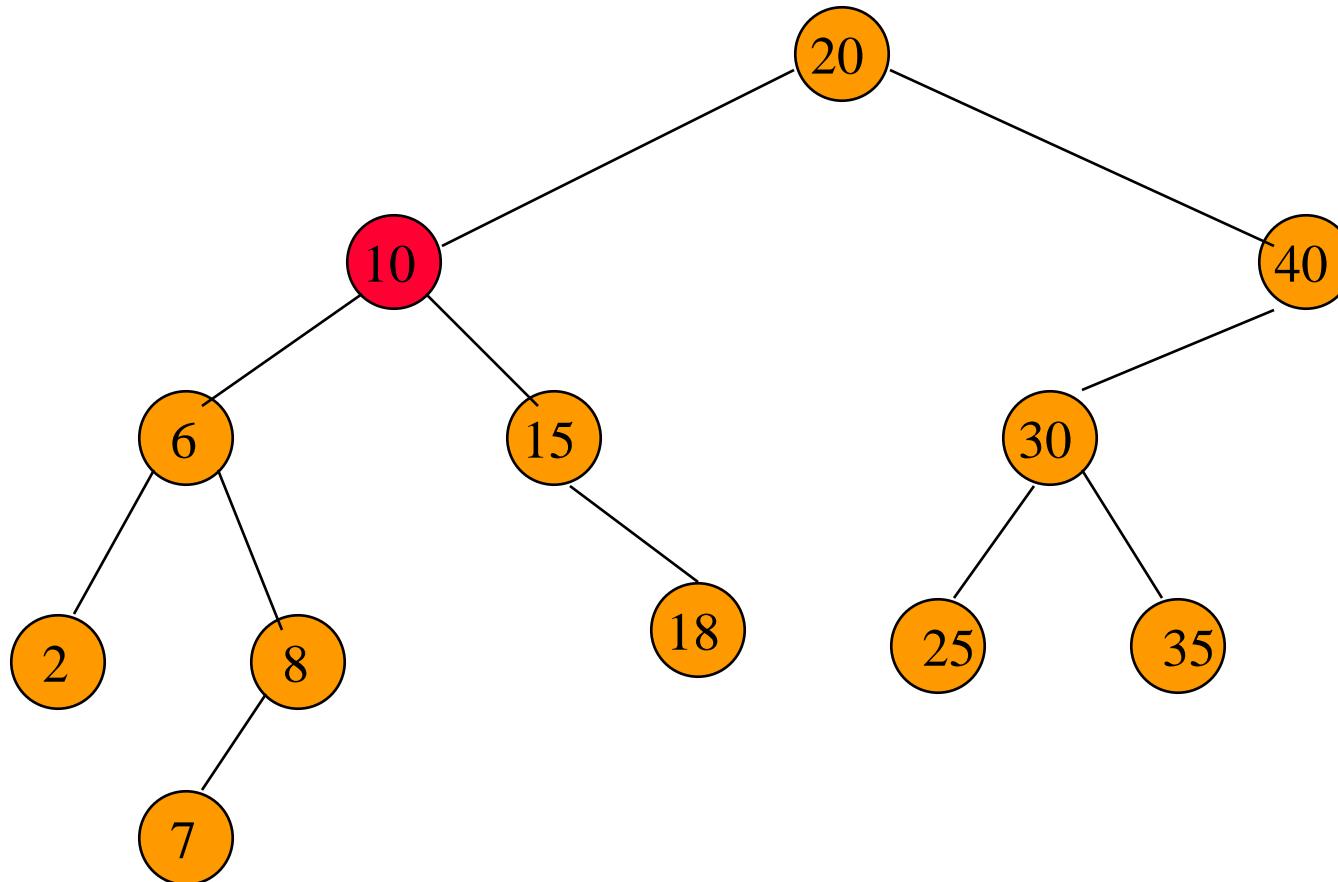
Remove from a degree 1 node. key = 15

Remove From A Degree 2 Node



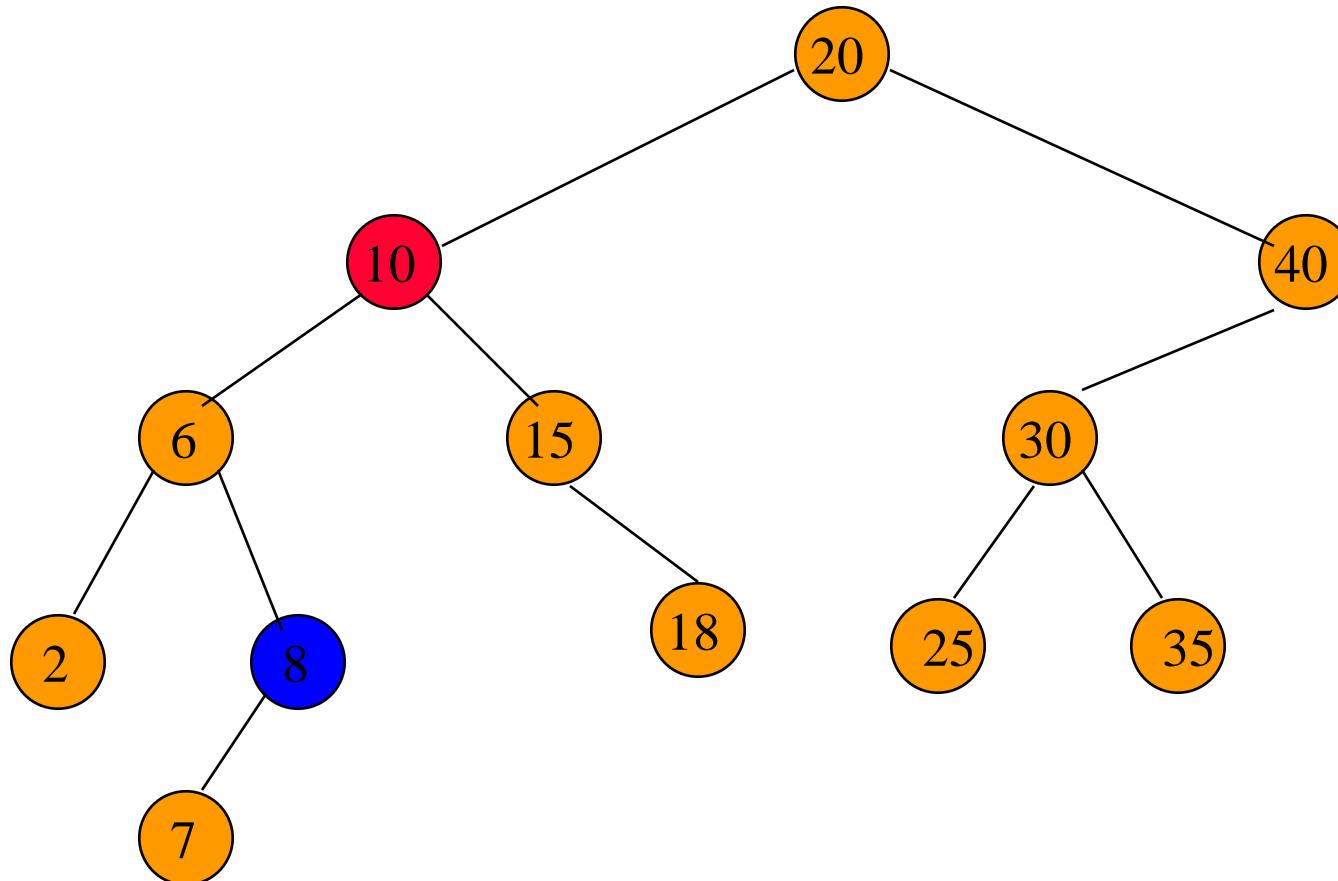
Remove from a degree 2 node. key = 10

Remove From A Degree 2 Node



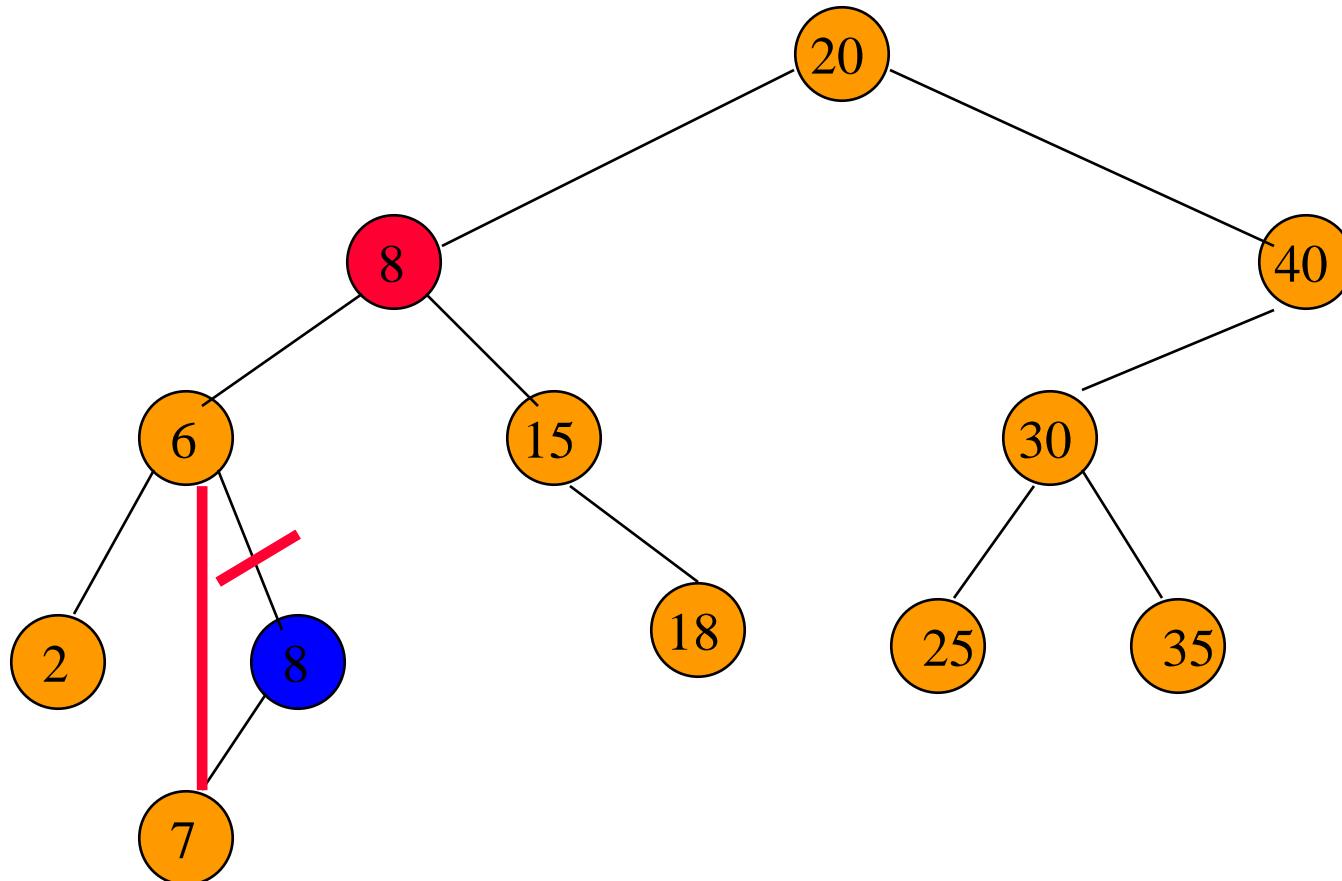
Replace with **largest** key in **left** subtree of the deleted node (or **smallest** in **right** subtree).

Remove From A Degree 2 Node



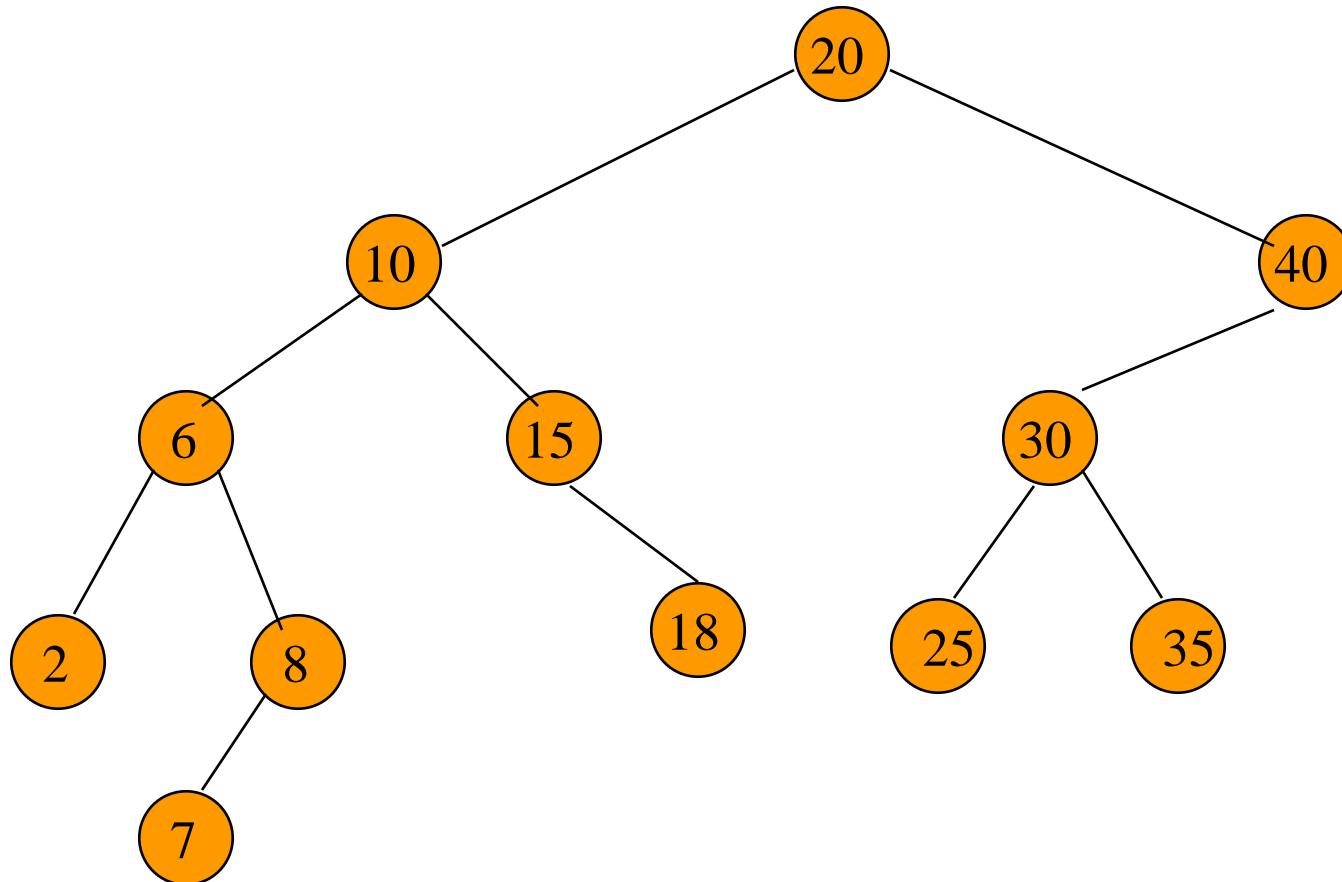
Replace with **largest** key in **left** subtree of the deleted node (or **smallest** in **right** subtree).

Remove From A Degree 2 Node



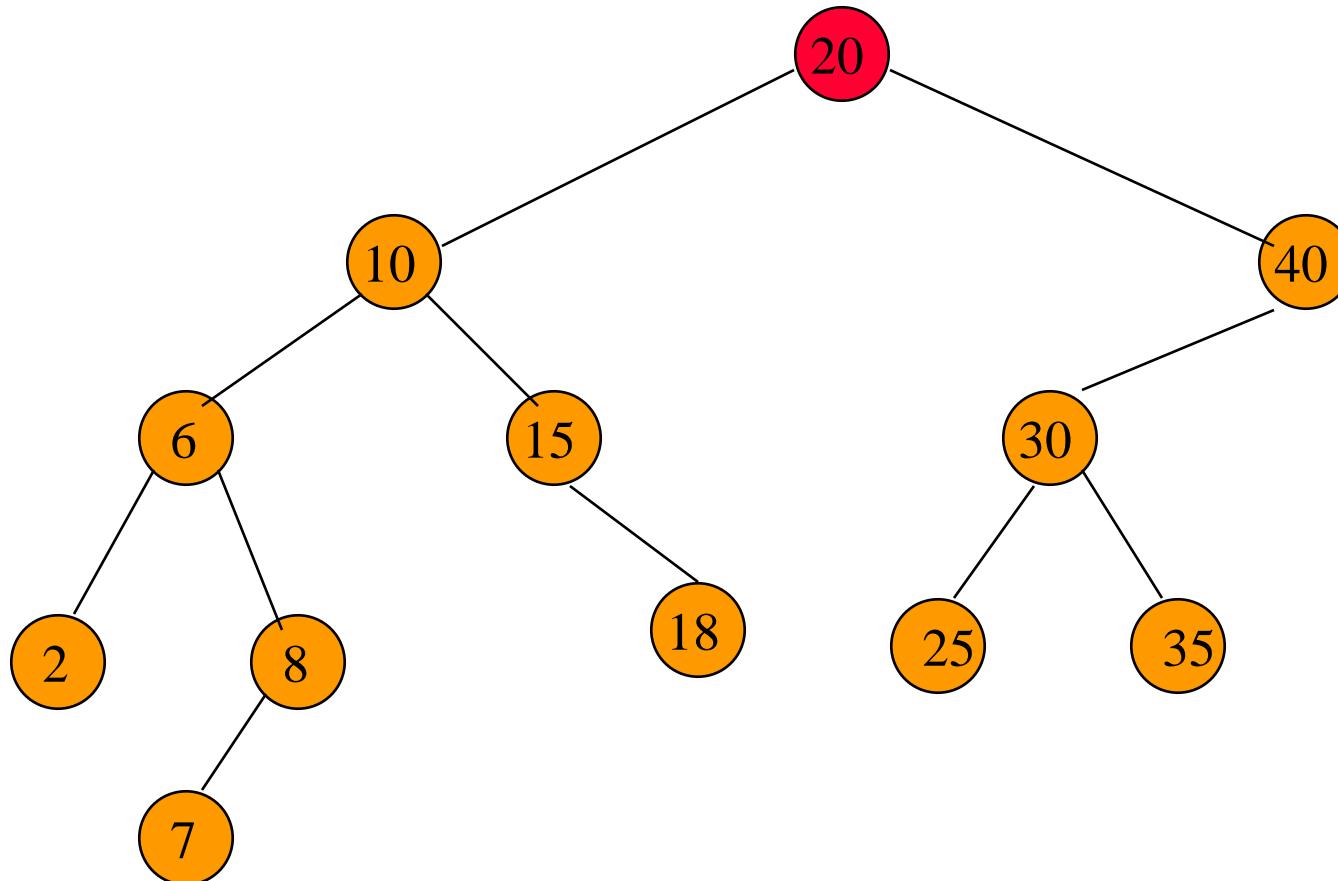
Largest key must be in a leaf or degree 1 node.

Another Remove From A Degree 2 Node



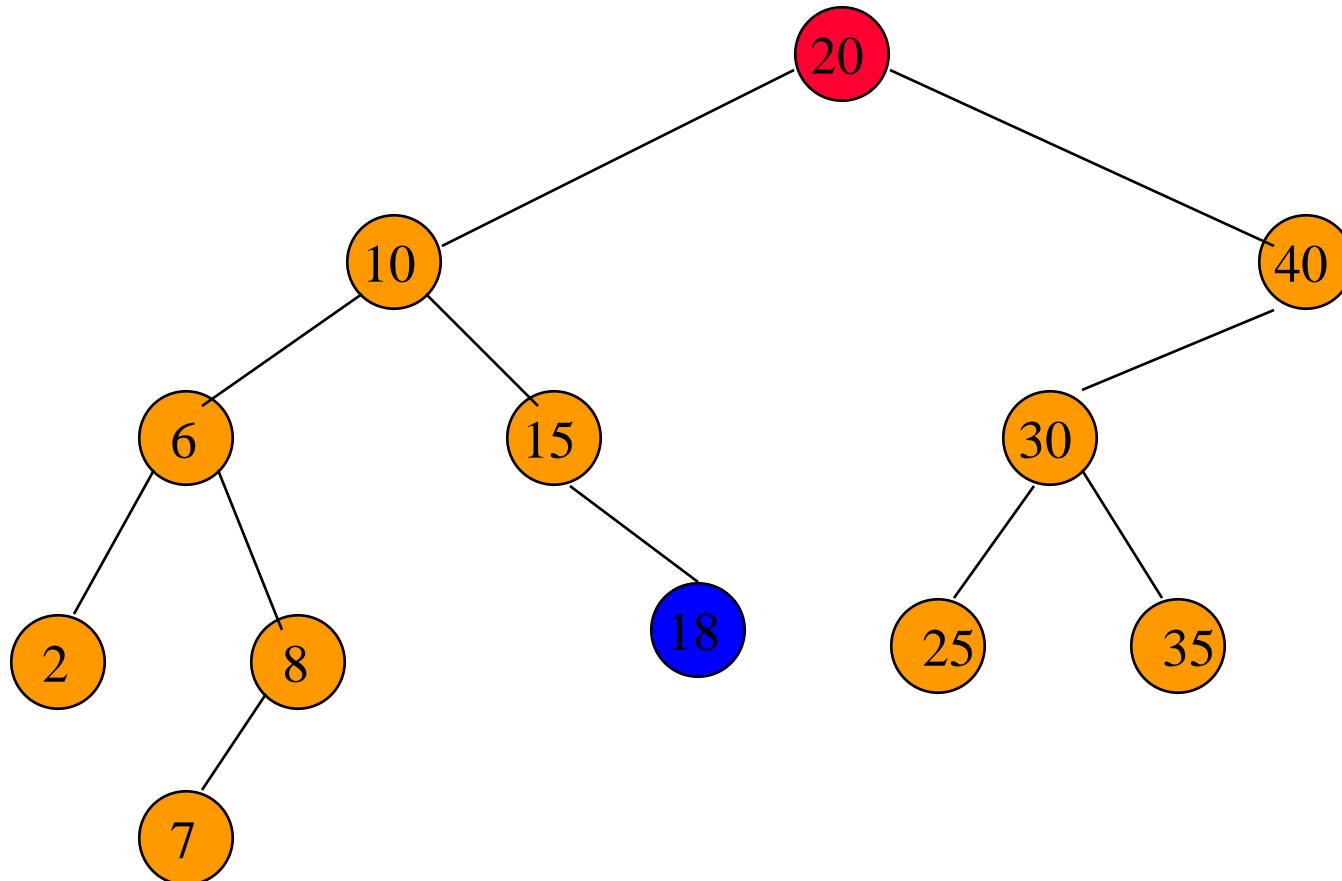
Remove from a degree 2 node. key = 20

Remove From A Degree 2 Node



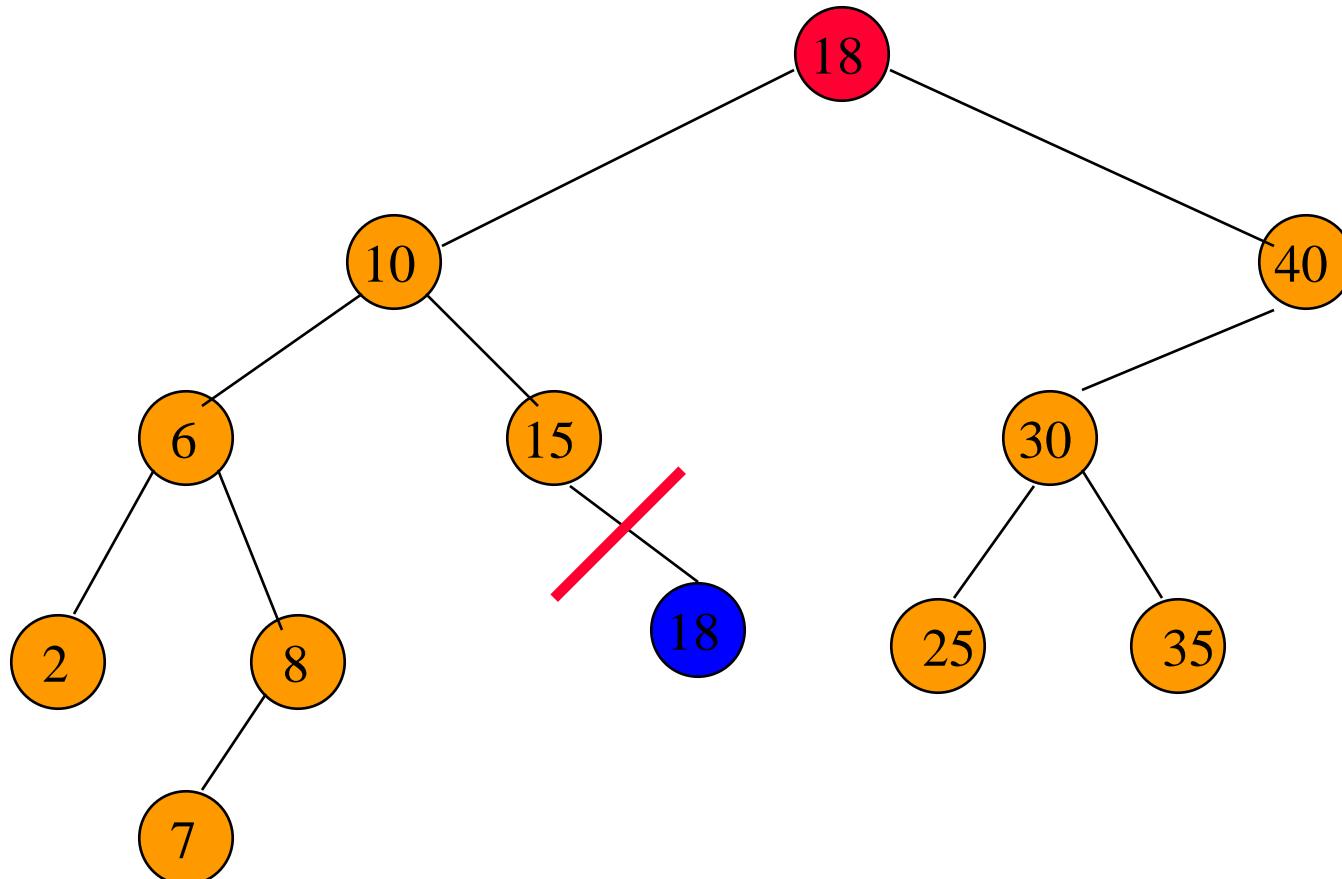
Replace with largest in left subtree of the deleted node.

Remove From A Degree 2 Node



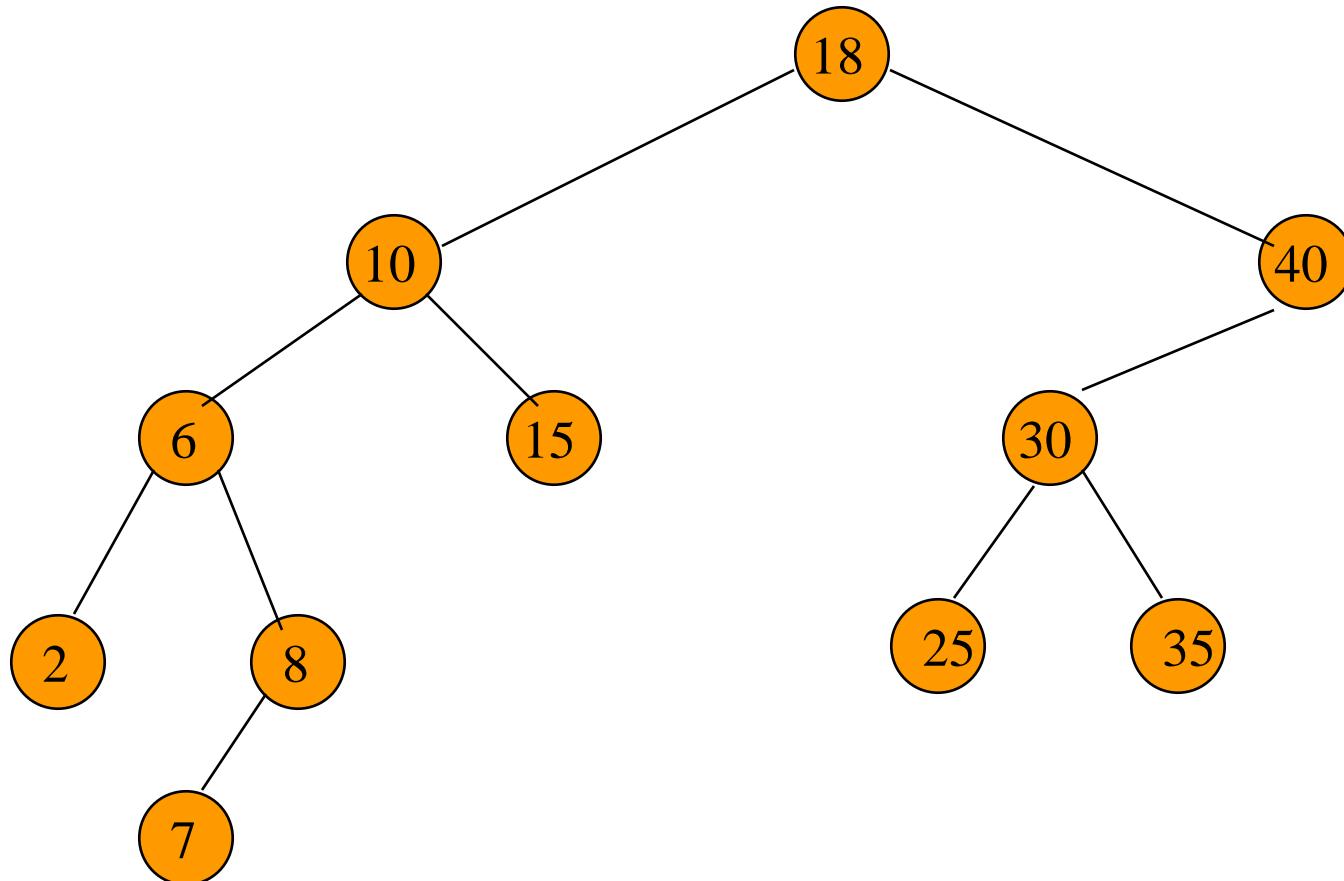
Replace with largest in left subtree of the deleted node.

Remove From A Degree 2 Node



Replace with largest in left subtree of the deleted node.

Remove From A Degree 2 Node



<https://www.youtube.com/watch?v=mtvbVLK5xDQ>
(2:00-6:00)



Find smallest

```
# To find the inorder successor which
# is the smallest node in the subtree

def findsuccessor(self, node):
    current_node = node
    while current_node.left != None:
        current_node = current_node.left
    return current_node
```

Remove()

```
def remove(self, root, key):  
    # Base Case  
    if root is None:  
        return root  
  
    # If the key to be deleted  
    # is smaller than the root's  
    # key then it lies in left subtree  
    if key < root.key:  
        root.left = self.remove(root.left, key)  
  
    # If the key to be deleted  
    # is greater than the root's key  
    # then it lies in right subtree  
    elif(key > root.key):  
        root.right = self.remove(root.right, key)
```

Remove()

```
# If key is same as root's key, then this is the node
# to be deleted
else:

    # Node with only one child or no child
    if root.left is None:
        temp = root.right
        root = None
        return temp

    elif root.right is None:
        temp = root.left
        root = None
        return temp
```

Remove()

```
# Node with two children:  
# Get the inorder successor  
# (smallest in the right subtree)  
temp = self.findsuccessor(root.right)  
  
# Copy the inorder successor's  
# content to this node  
root.key = temp.key  
  
# Delete the inorder successor  
root.right = self.remove(root.right, temp.key)  
  
return root
```

Balanced BST / AVL Tree



Balanced Binary Search Trees

- AVL (Adel'son-Vel'skii and Landis) trees
- Hopefully, height is $O(\log_2 n)$, where n is the number of elements in the tree
- Ideally, **get**, **put**, and **remove** take $O(\log_2 n)$ time

AVL Tree

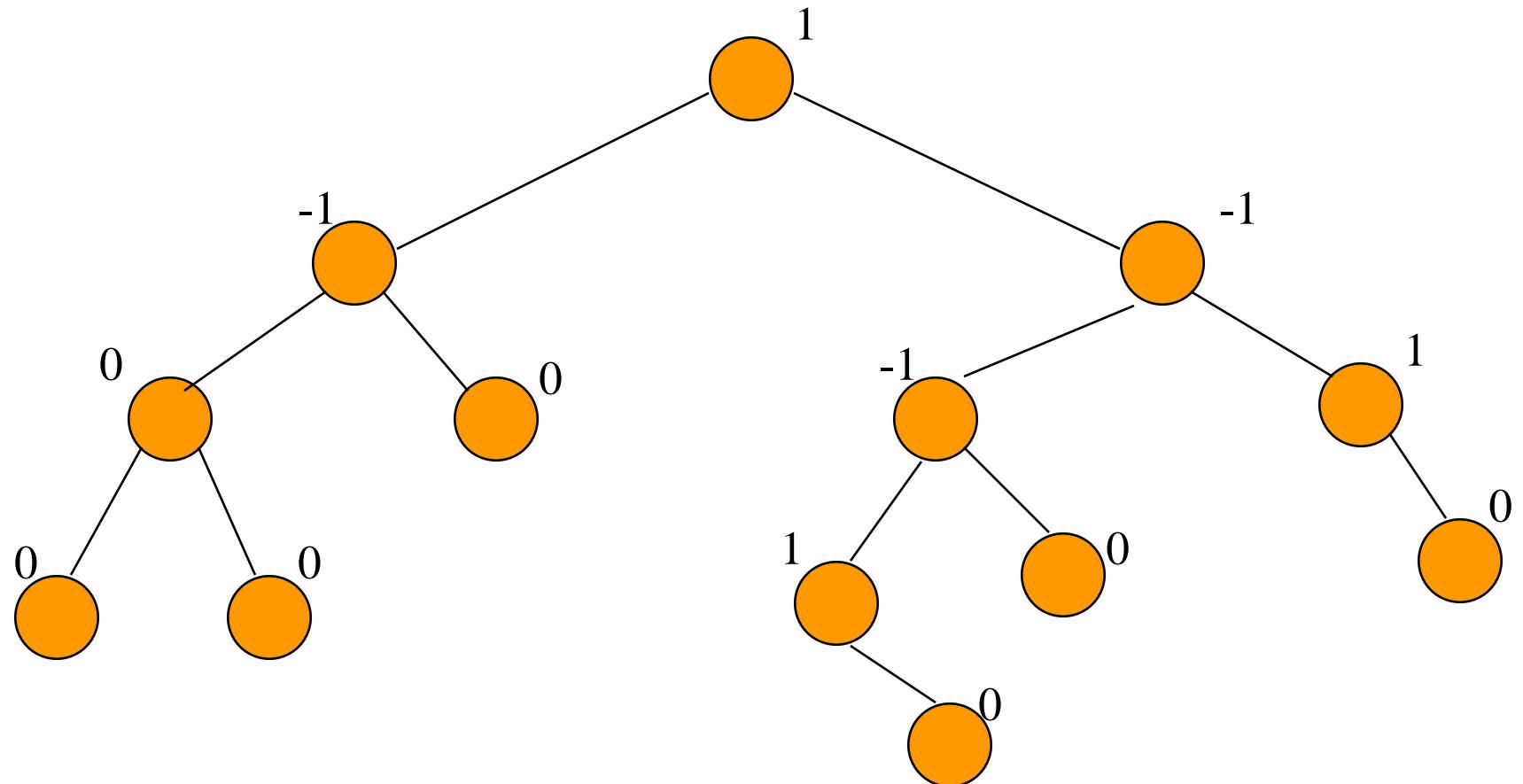
- binary tree
- for every node x , define its balance factor
 $\text{balance factor of } x = \text{height of right subtree of } x - \text{height of left subtree of } x$
- balance factor of every node x is $-1, 0$, or 1

Note: In some texts (e.g. the textbook by Sahni), the balance factor is computed as follows:

balance factor of $x = \text{height of left subtree of } x - \text{height of right subtree of } x$

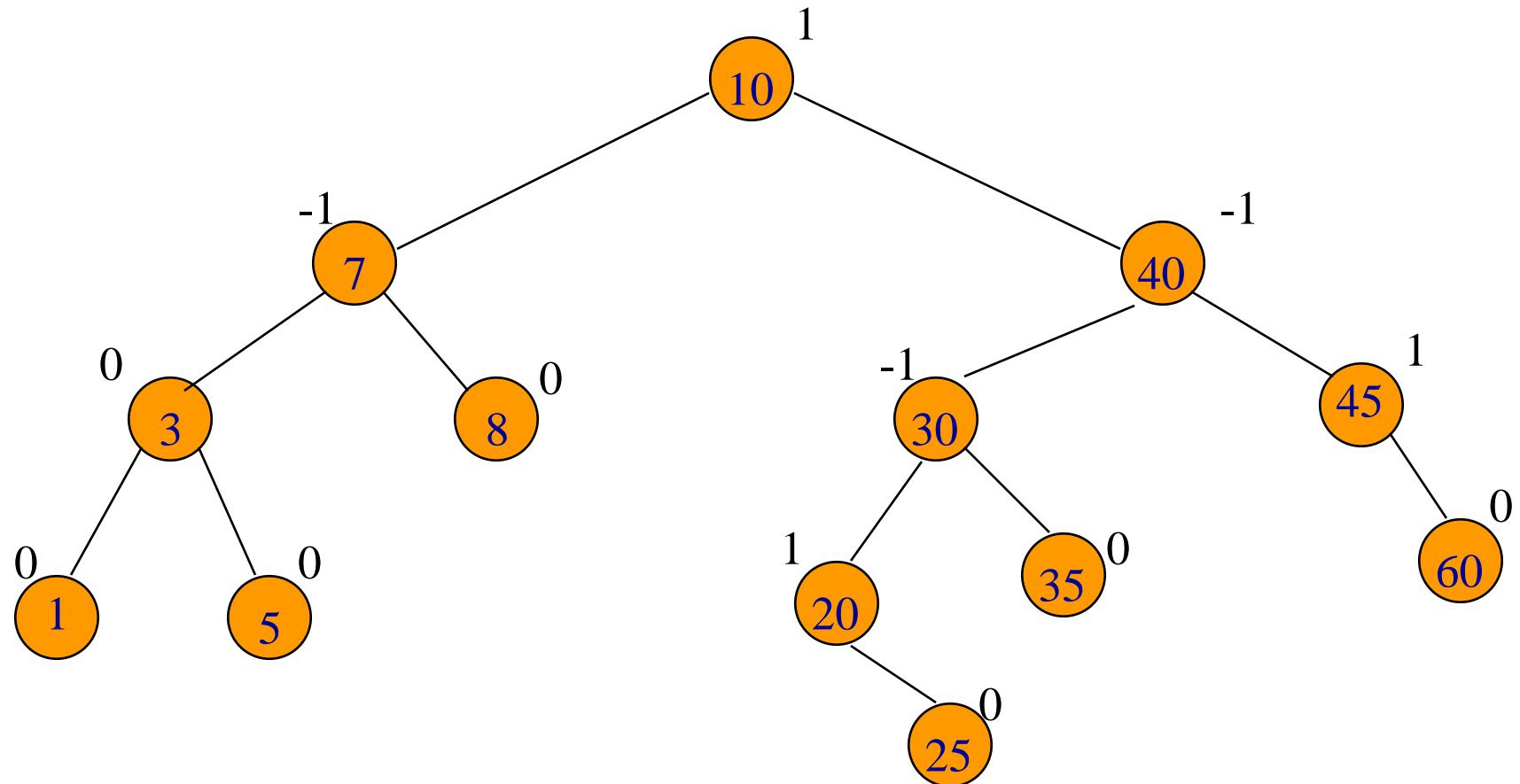
This note uses the above-mentioned implementation.

Balance Factors

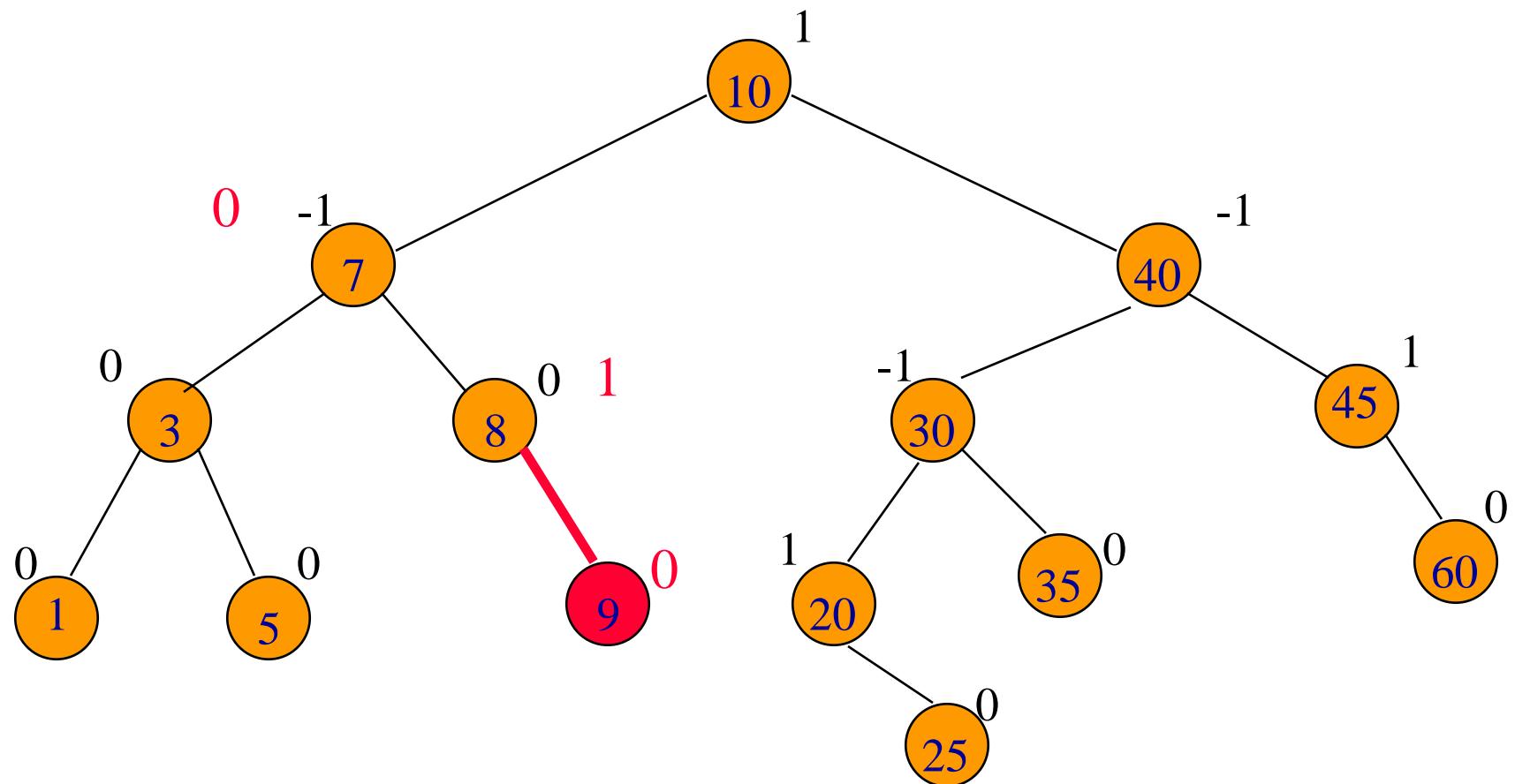


This is an AVL tree.

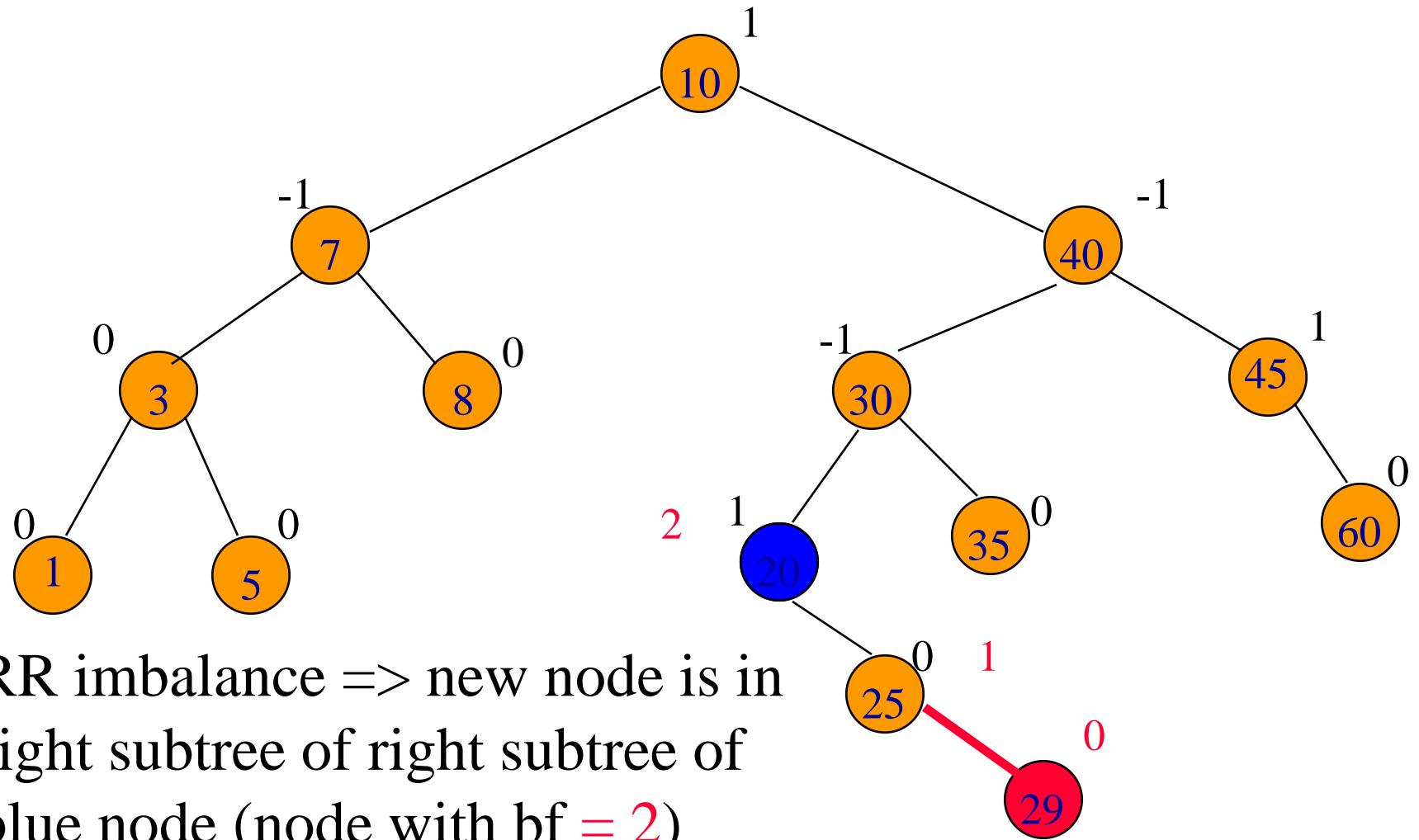
AVL Search Tree



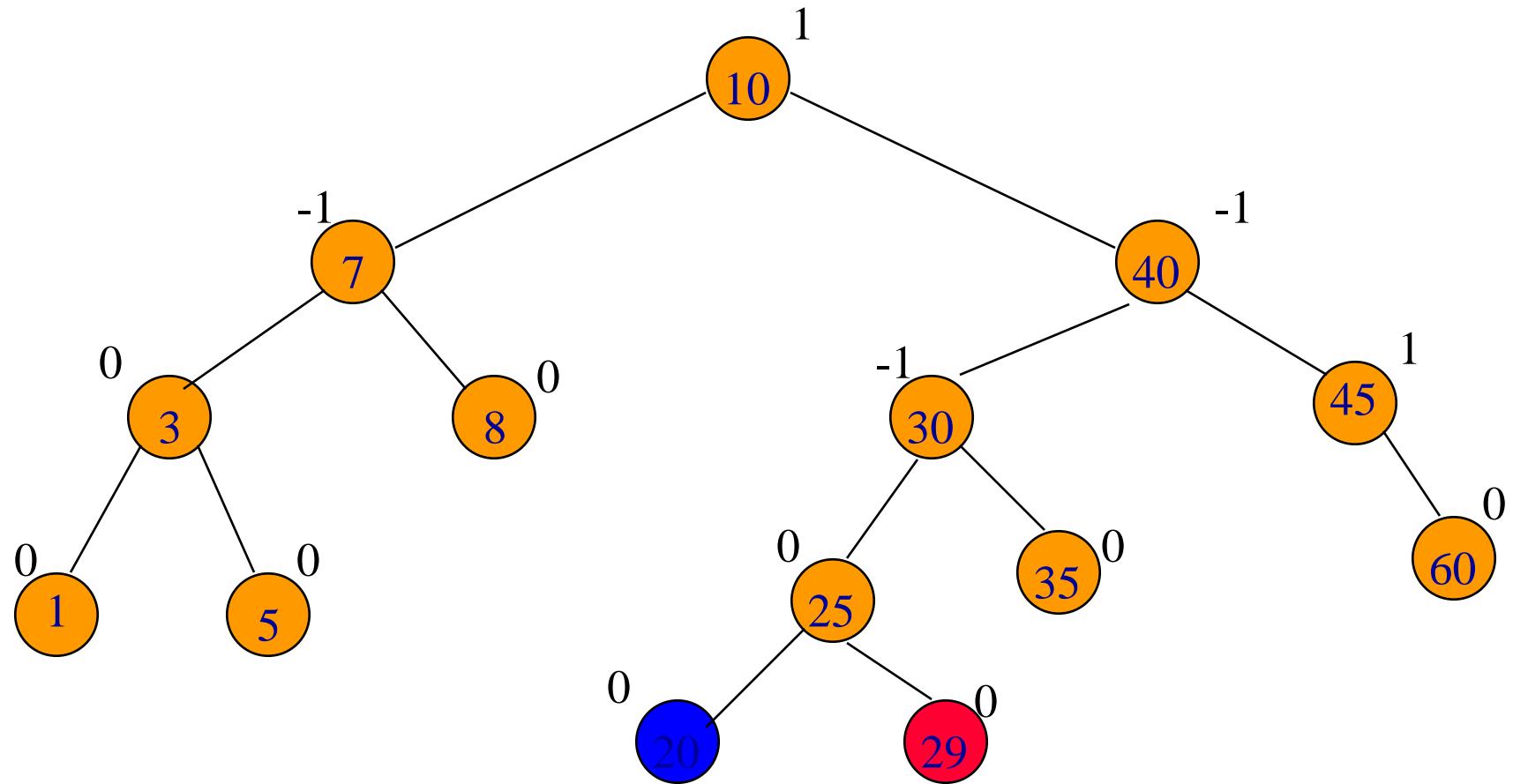
$\text{put}(9)$



put(29)



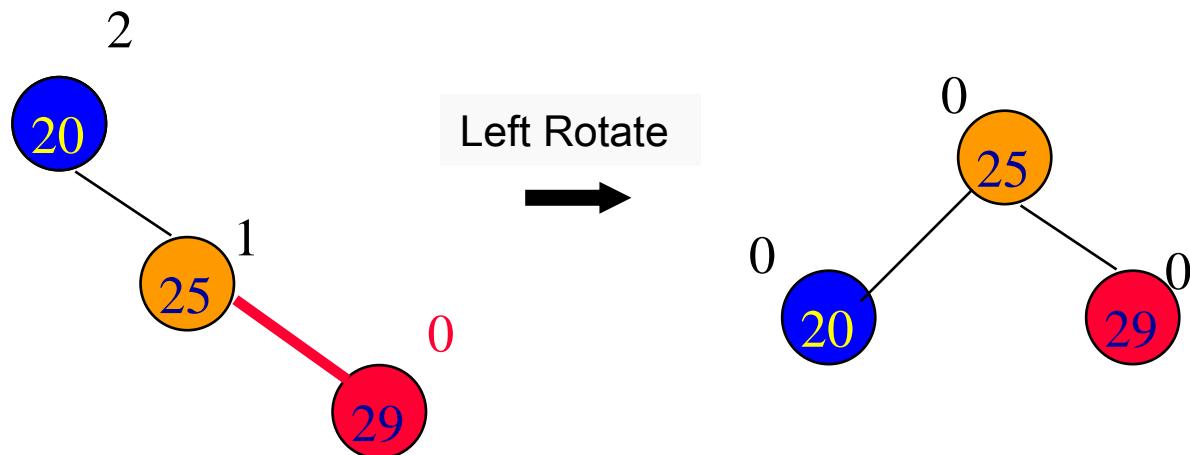
put(29)



RR rotation.

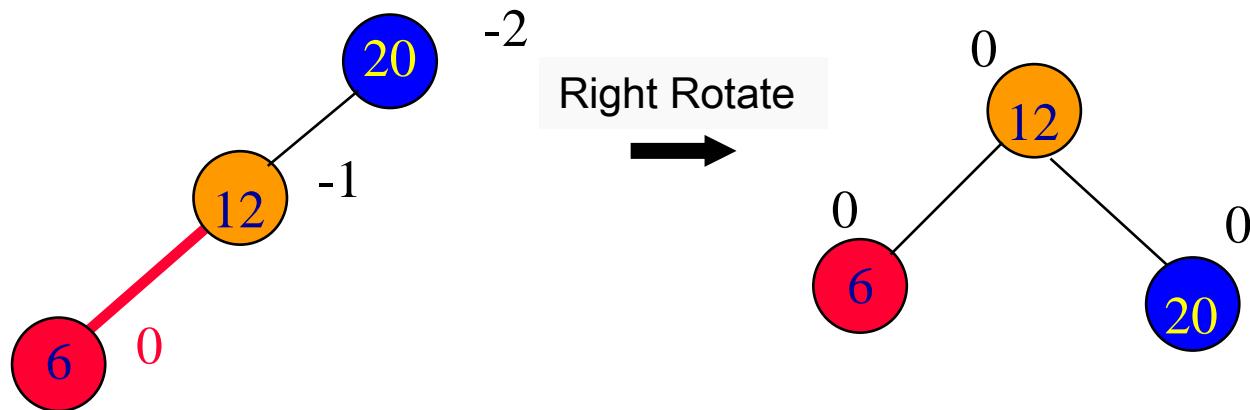
AVL Rotations

- RR



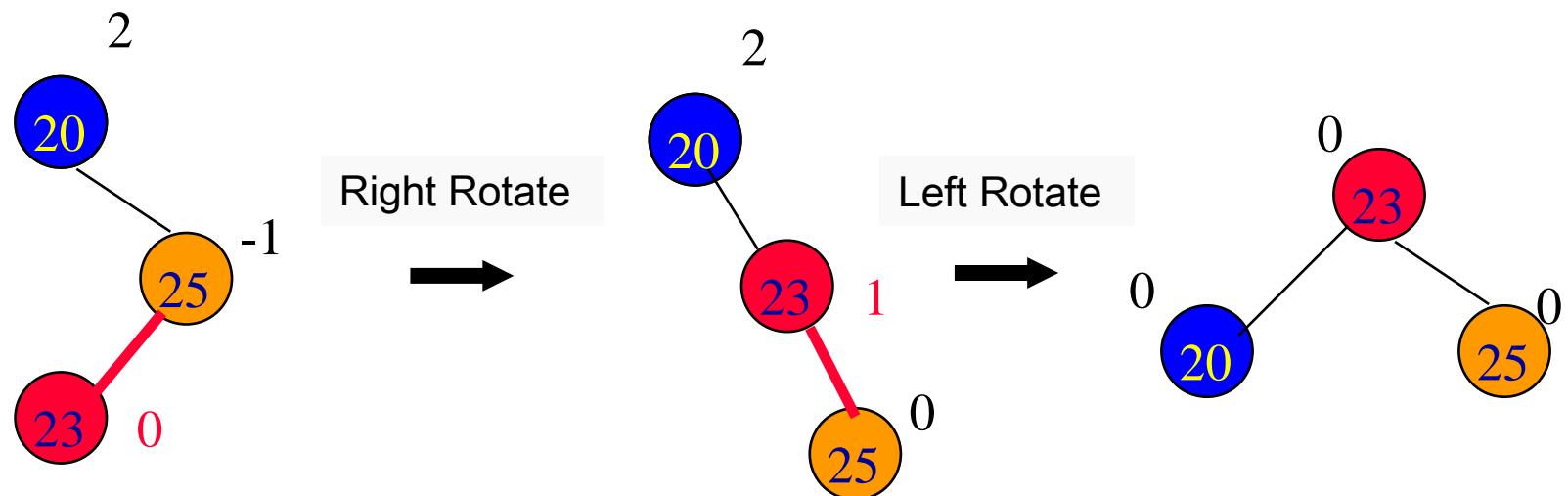
AVL Rotations

- LL



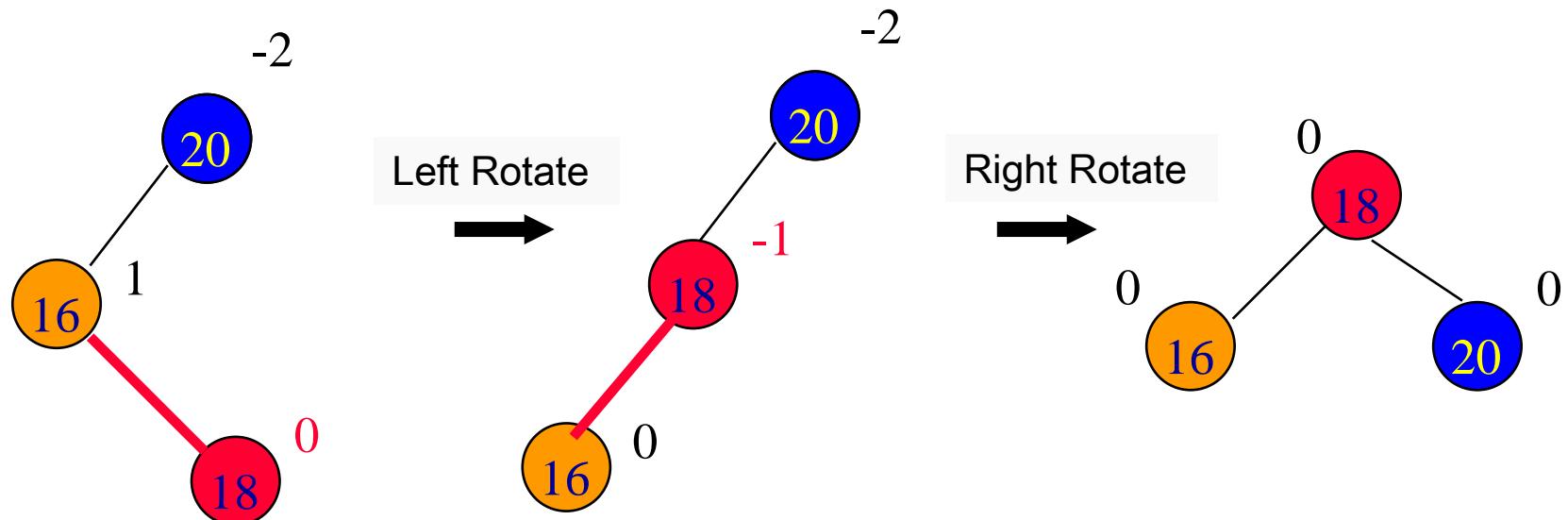
AVL Rotations

- RL



AVL Rotations

- LR



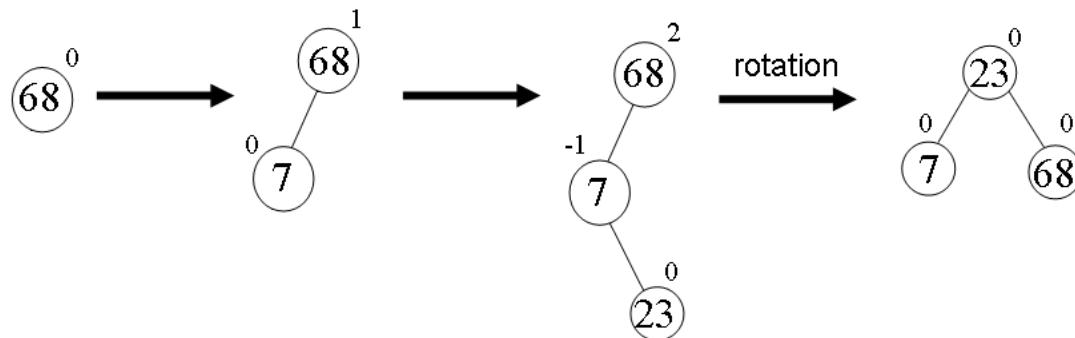
<https://www.youtube.com/watch?v=yAFLlCZFJy0>



Exercise

balance factor of node i = height of left subtree of node i -
height of right subtree of node i

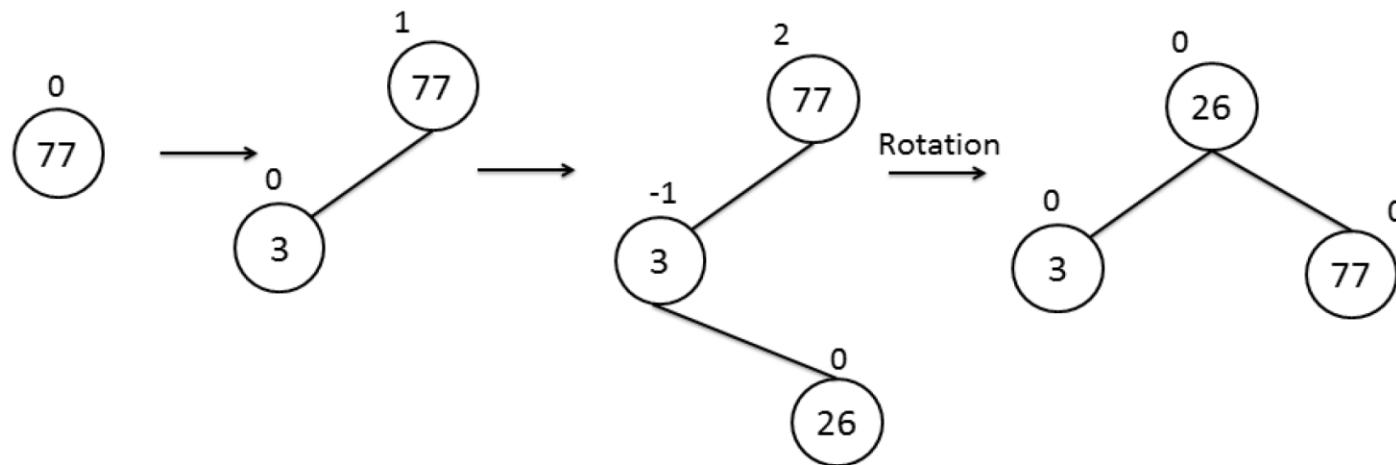
**the sequence of integer keys 45, 3, 12, 52 is to be inserted
into the AVL tree**



Exercise

Suppose the sequence of integer keys 77, 3, 26 is to be inserted into an AVL tree based on the following balance factor for every node i of an AVL tree:

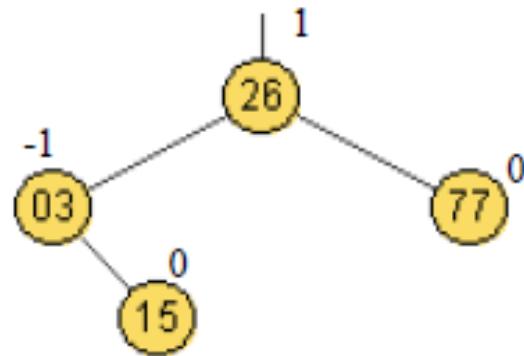
$$\text{balance factor of node } i = \text{height of left subtree of node } i - \text{height of right subtree of node } i$$



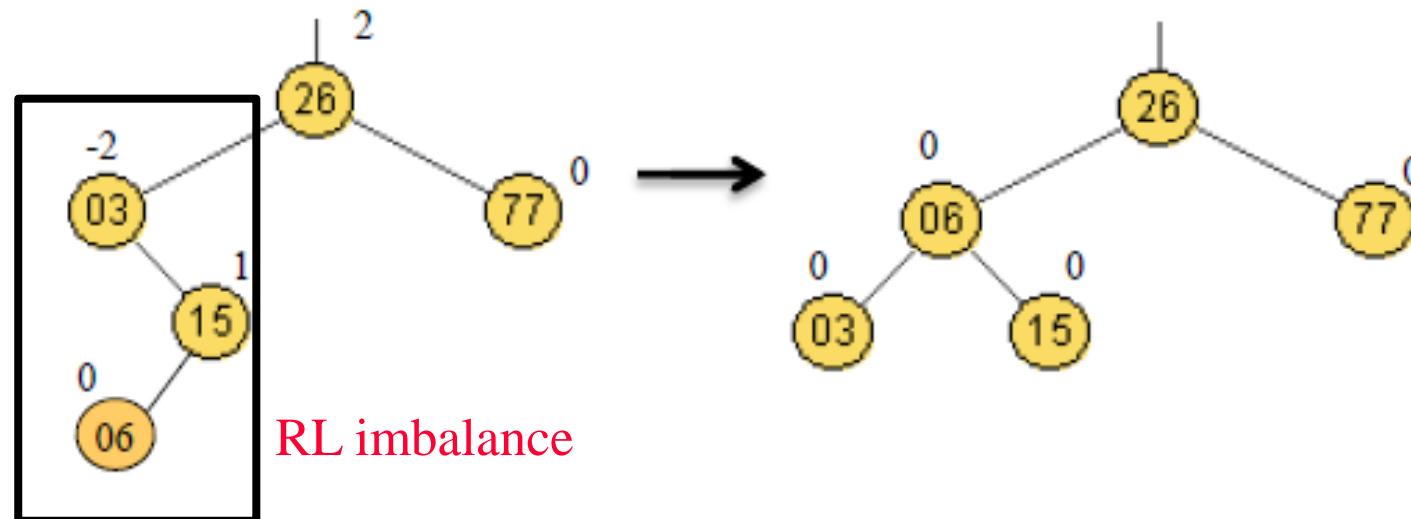
Above figure shows the keys with the balance factors of the AVL trees and necessary rotation for each insertion. Then, the sequence of integer keys 15, 6, 22 is to be inserted into the AVL tree with the keys 77, 3 and 26 in above figure. Draw the AVL tree, indicate the keys, balance factors and rotation if any after **EACH** insertion for the sequence of integer keys 15, 6, 22.

(10 marks)
54

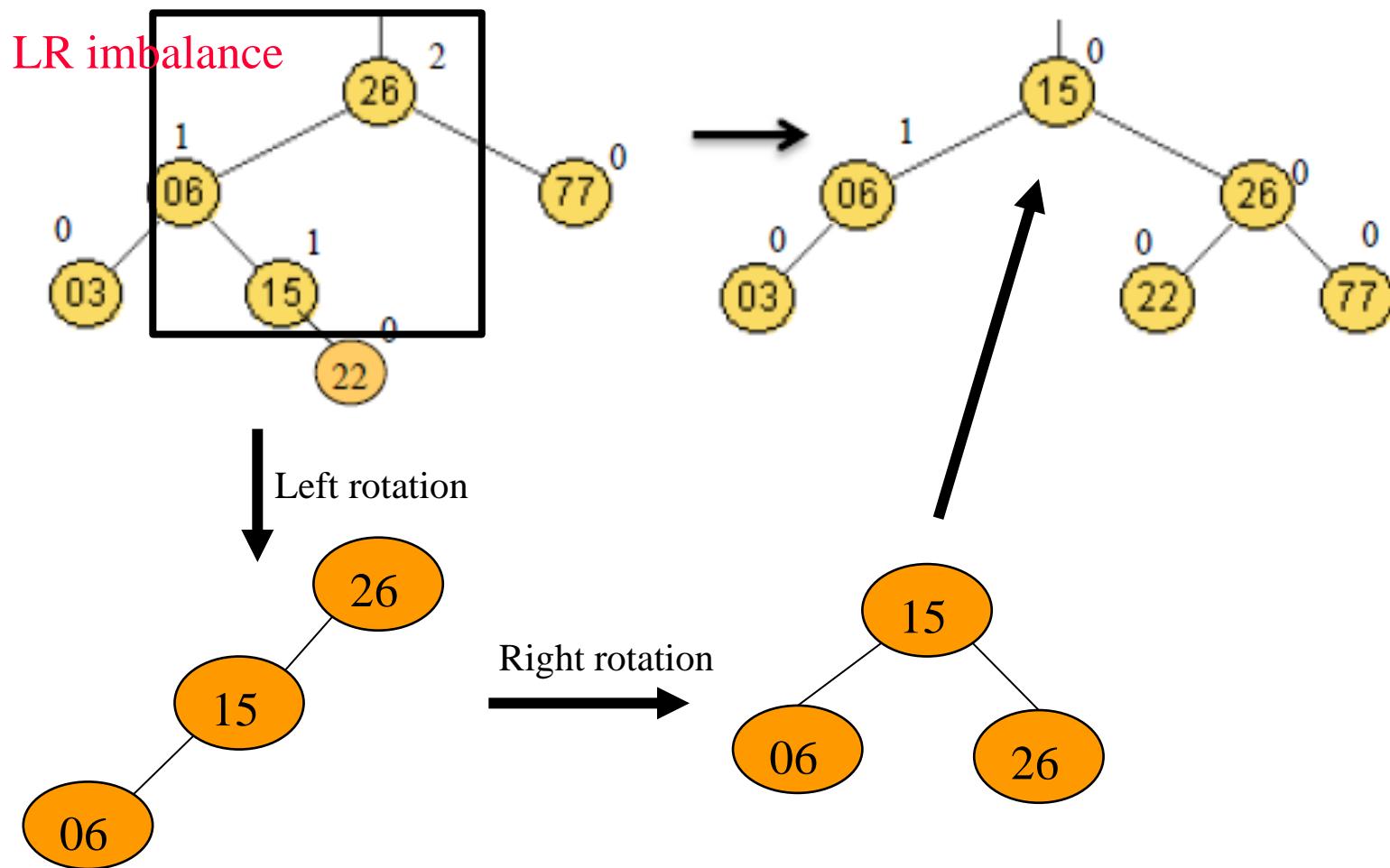
insert 15



insert 6 with rotation



insert 22 with rotation



Summary

- Binary Search Tree
- Operations
- AVL

SEHH2239 Data Structures

Lecture 10

Learning Objectives:

- To describe the Min / Max Trees
- To understand Heap structure and implement it in array
- To put and remove items in Heap
- To convert data to heap by heapifying and its performance
- To implement the priority queue
- To sort data by heapsort

Heap and Heap Sort



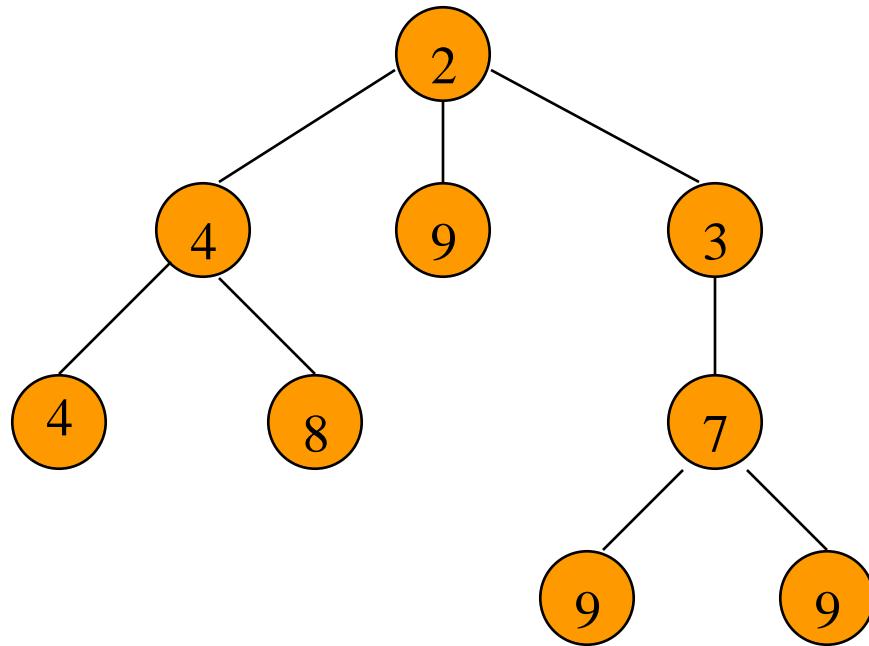
Two kinds of heap:

- Min Heap.
- Max Heap.

Min Tree Definition

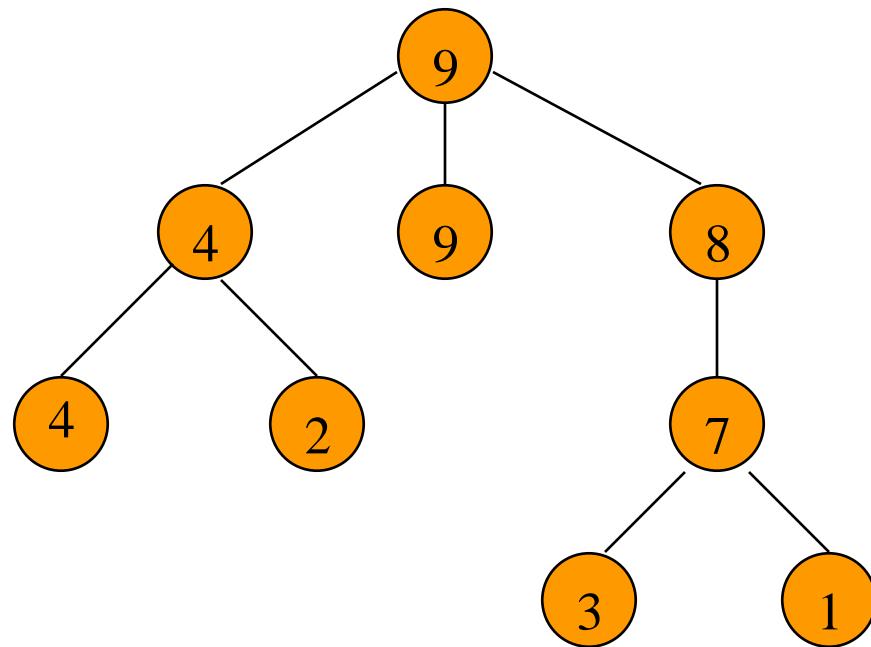
- Each tree node has a value.
- Value in any node is the **minimum value** in the subtree for which that node is the **root**.
- Equivalently, no descendent has a smaller value.

Min Tree Example



Root has minimum element.

Max Tree Example

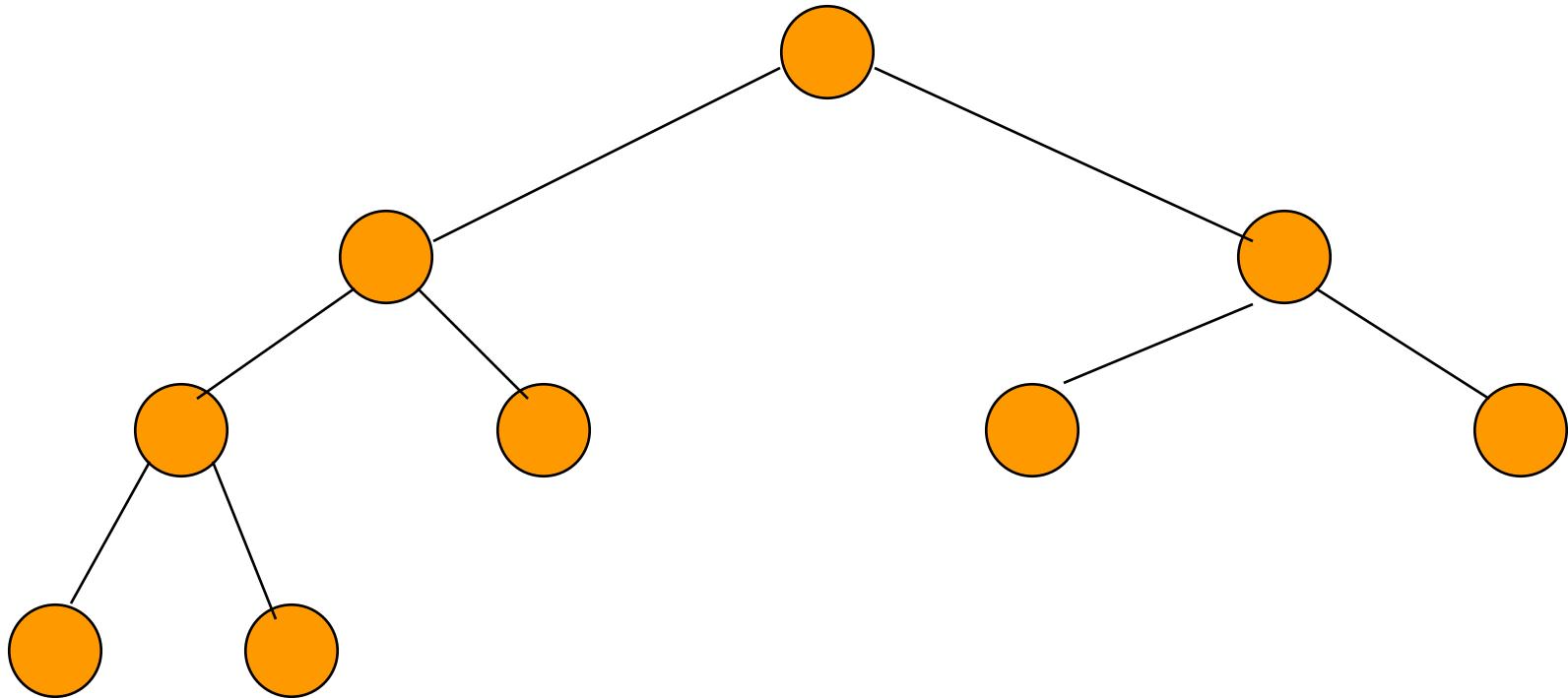


Root has maximum element.

Min Heap Definition

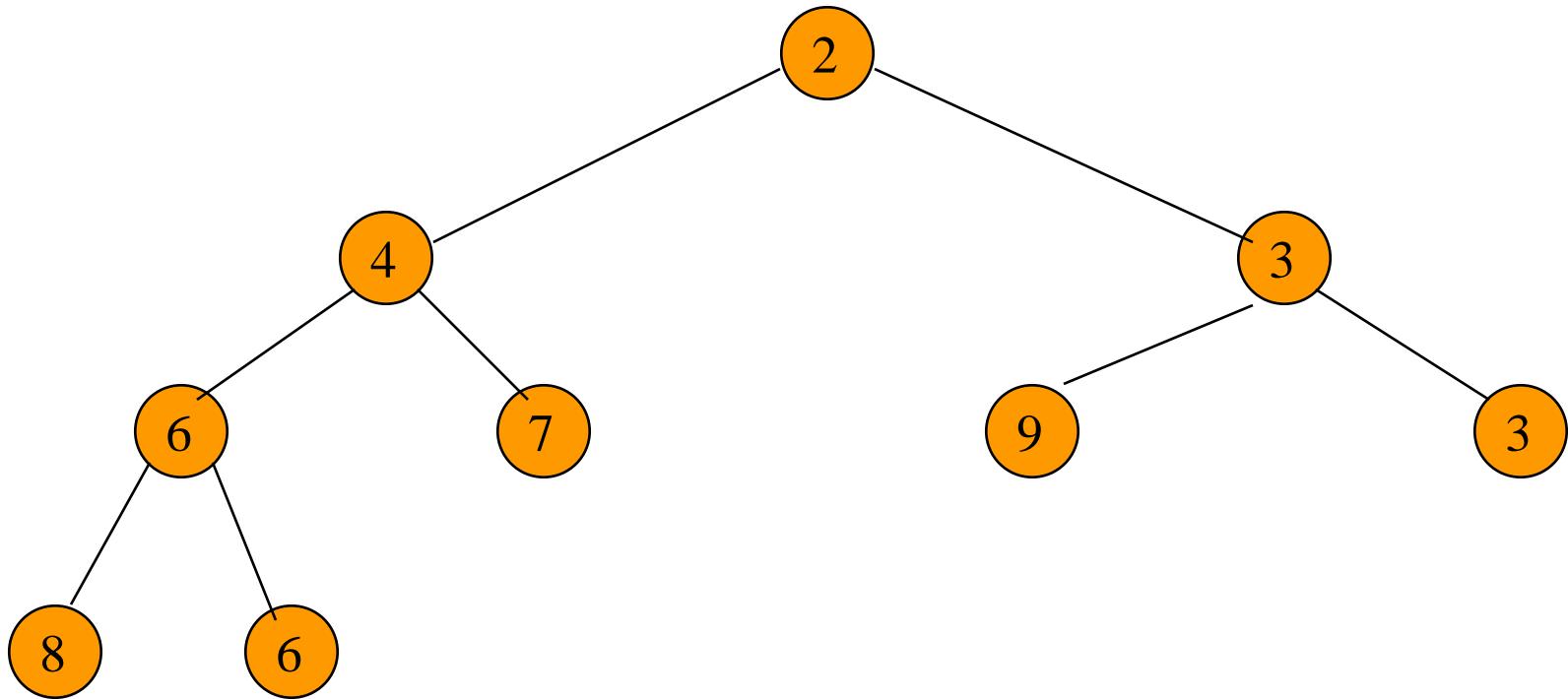
- **complete** binary tree
- **min** tree

Min Heap With 9 Nodes



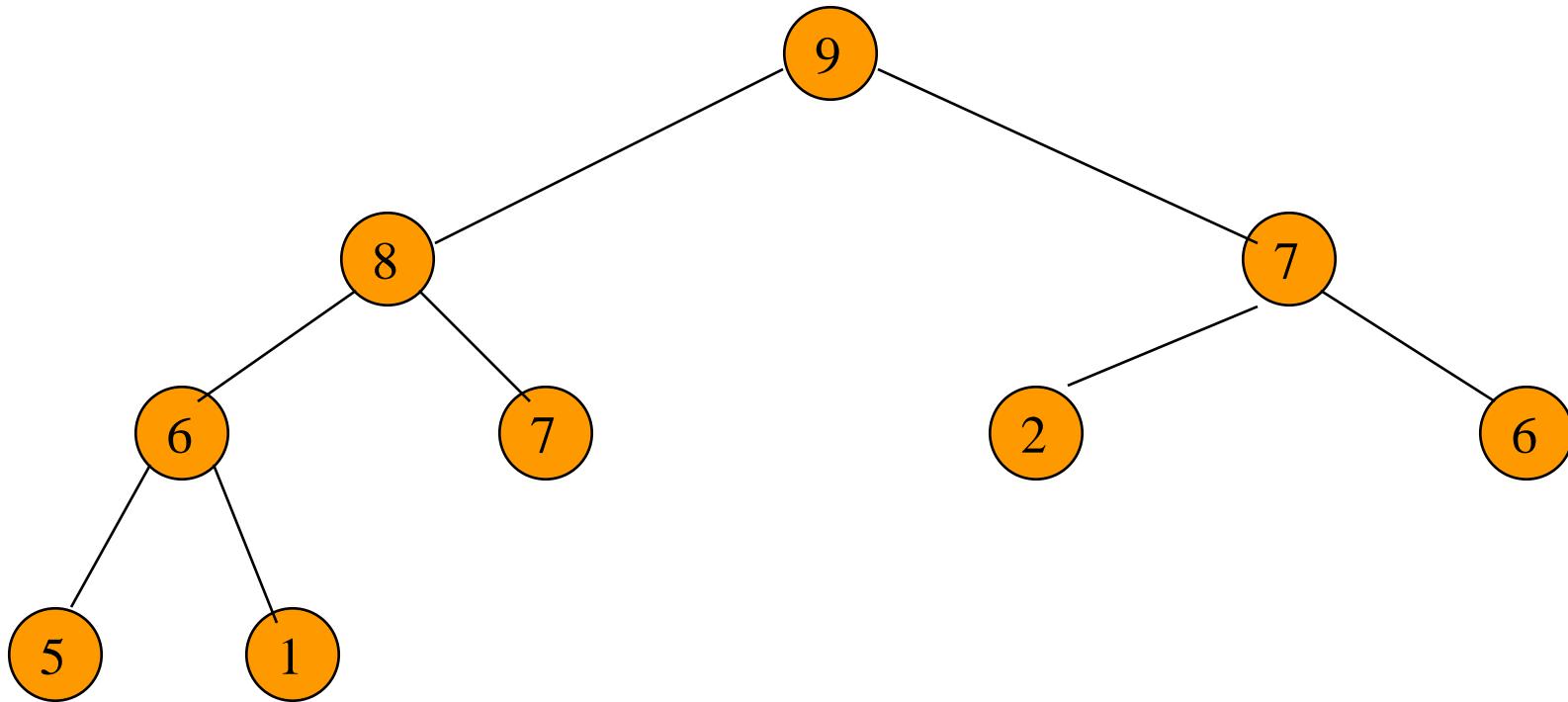
Complete binary tree with 9 nodes.

Min Heap With 9 Nodes



Complete binary tree with 9 nodes
that is also a min tree.

Max Heap With 9 Nodes



Complete binary tree with 9 nodes
that is also a max tree.

Heap Height

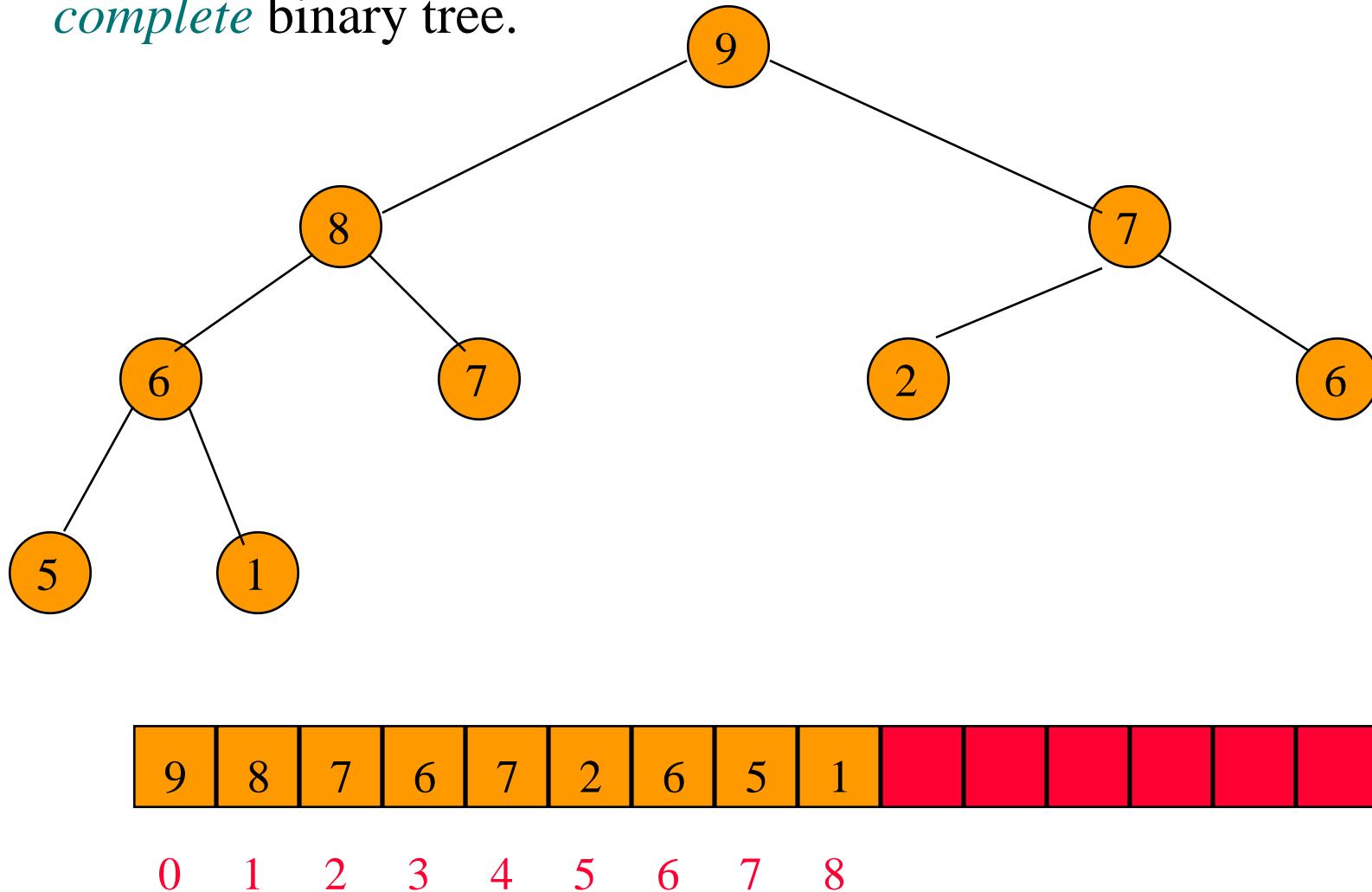
- By default, a **heap** is a **max heap**; a **min heap** must be explicitly specified.
- Since a heap is a complete binary tree, the height of an **n** node heap is

$$\lfloor \log_2 n \rfloor + 1$$

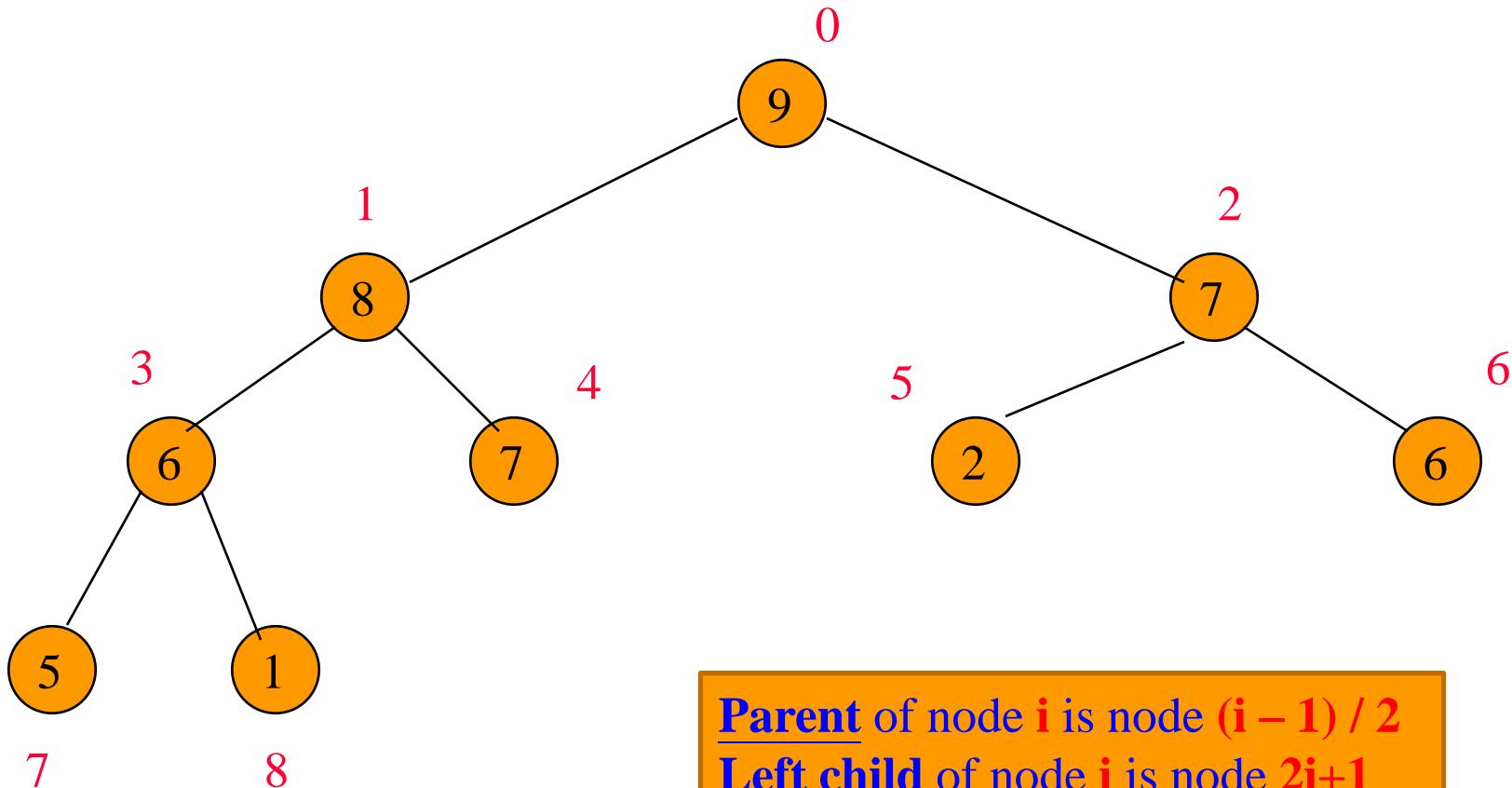
Heap implemented in An Array

A Heap Represented As An Array

- A Heap Is Efficiently Represented in an array as a *complete* binary tree.



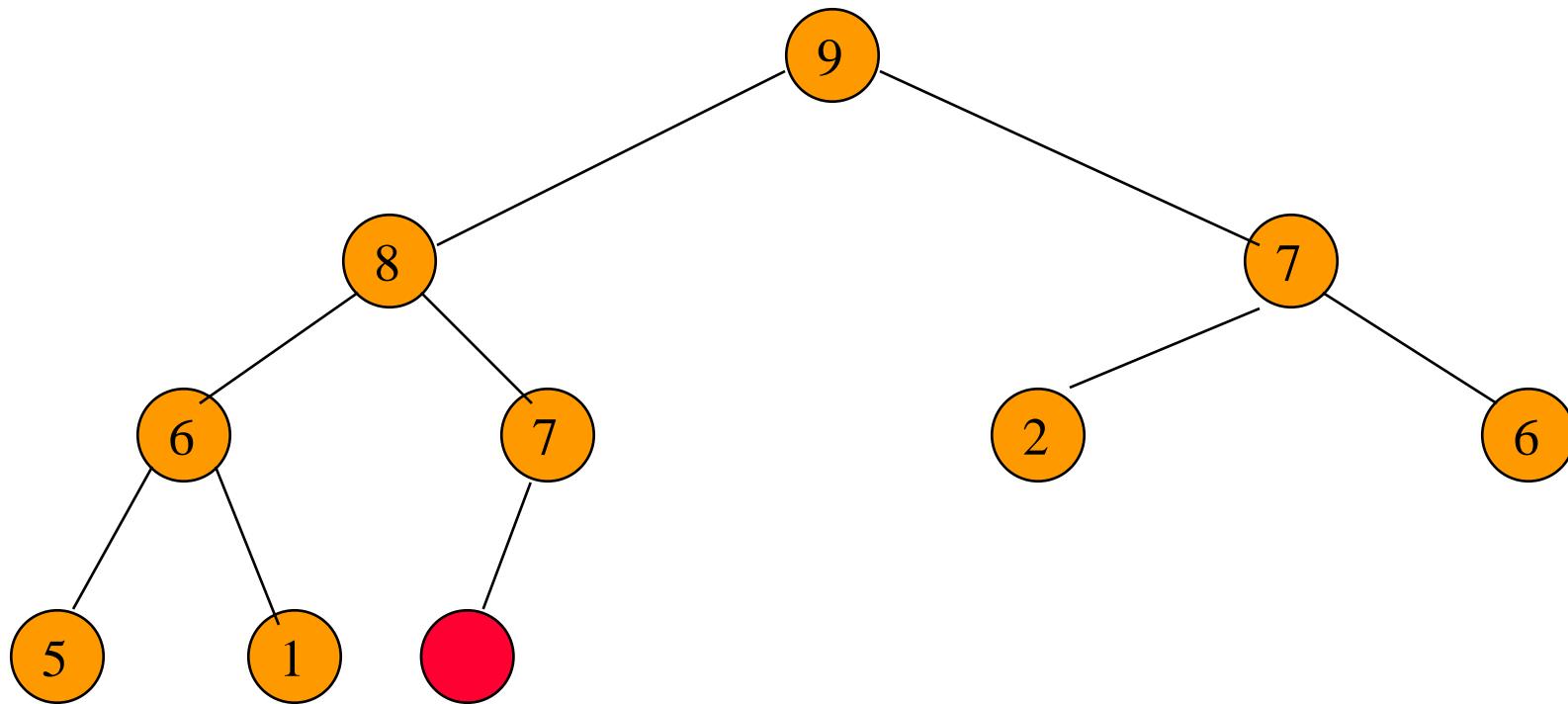
Moving Up And Down A Heap



Parent of node i is node $(i - 1) / 2$
Left child of node i is node $2i+1$
Right child of node i is node $2i+2$

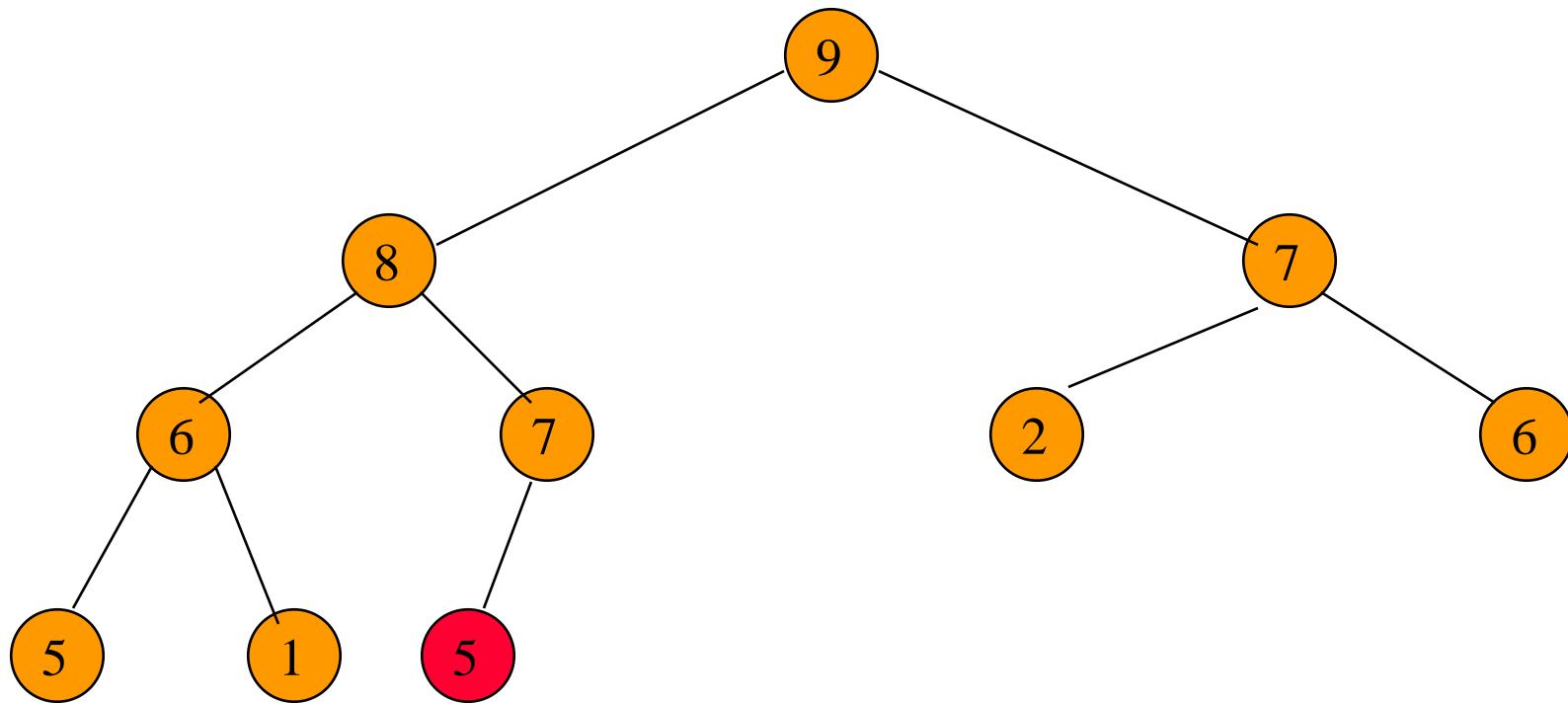
Implementation of Heap - PUT

Putting An Element Into A Max Heap



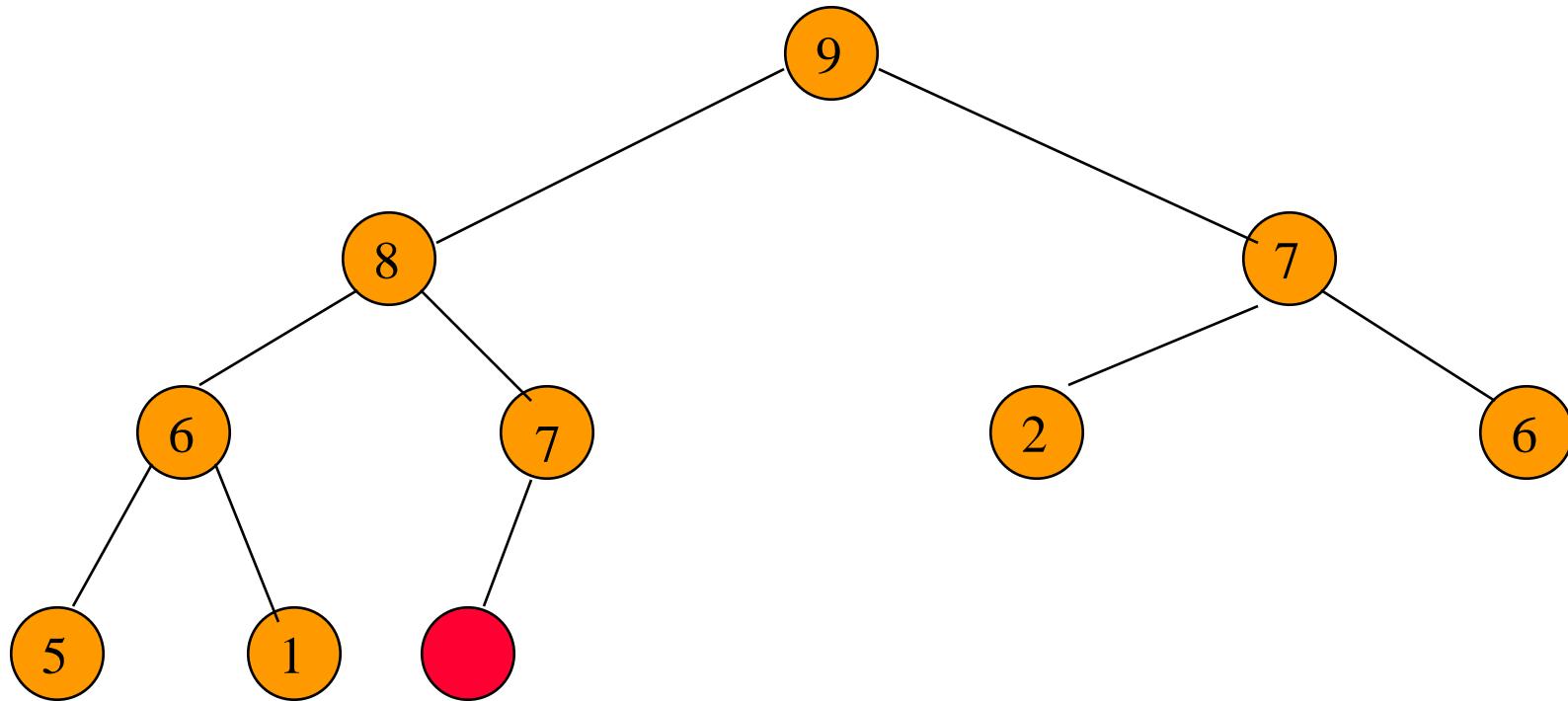
Complete binary tree with 10 nodes.

Putting An Element Into A Max Heap



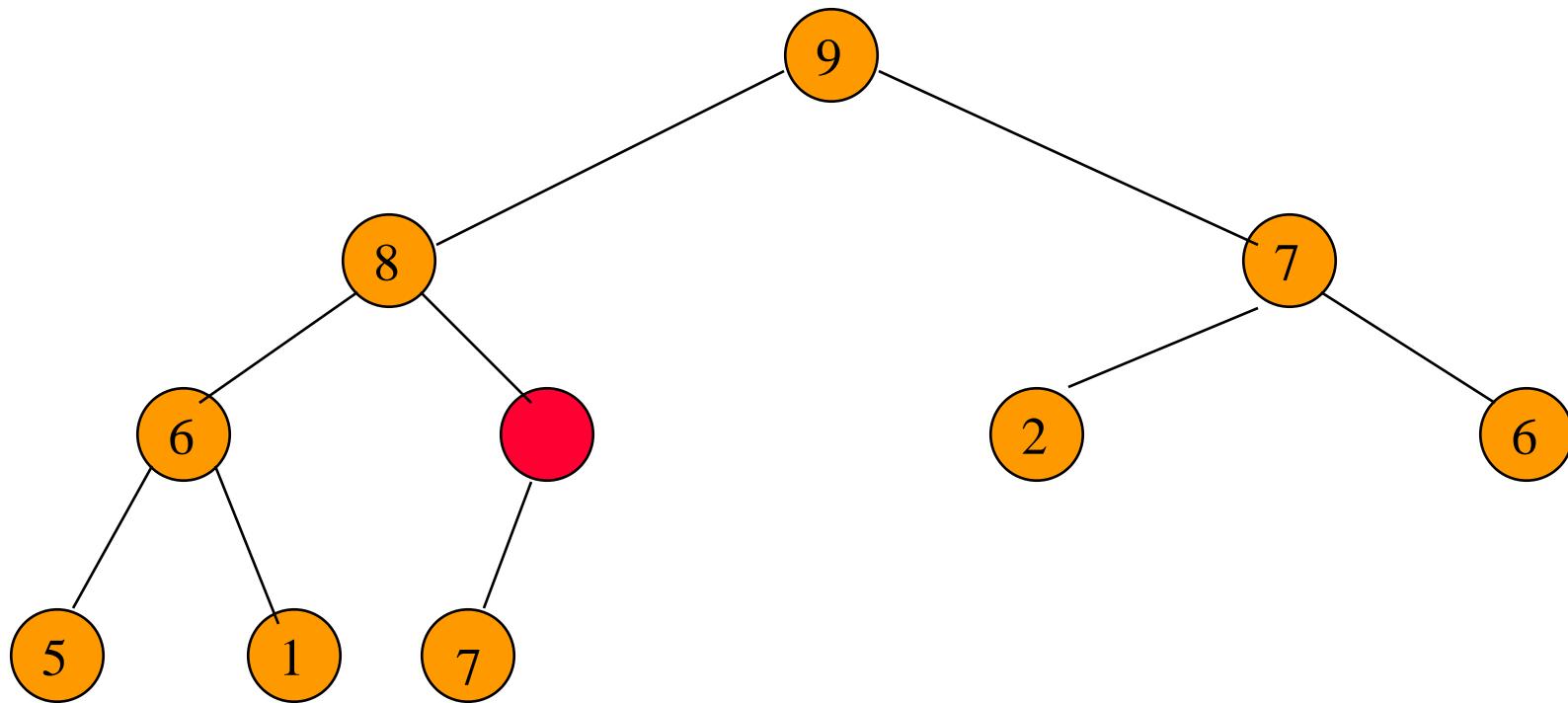
New element is 5.

Putting An Element Into A Max Heap



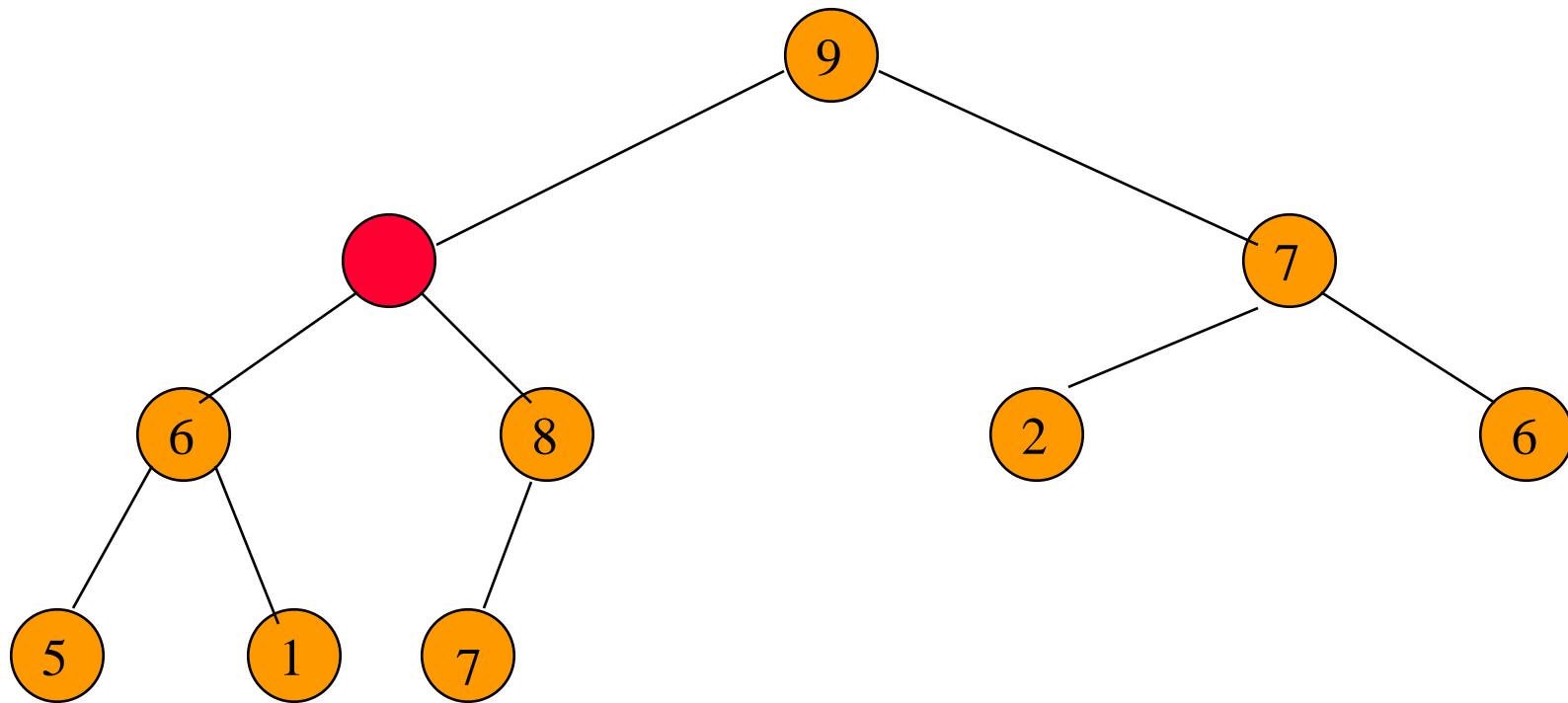
New element is 20.

Putting An Element Into A Max Heap



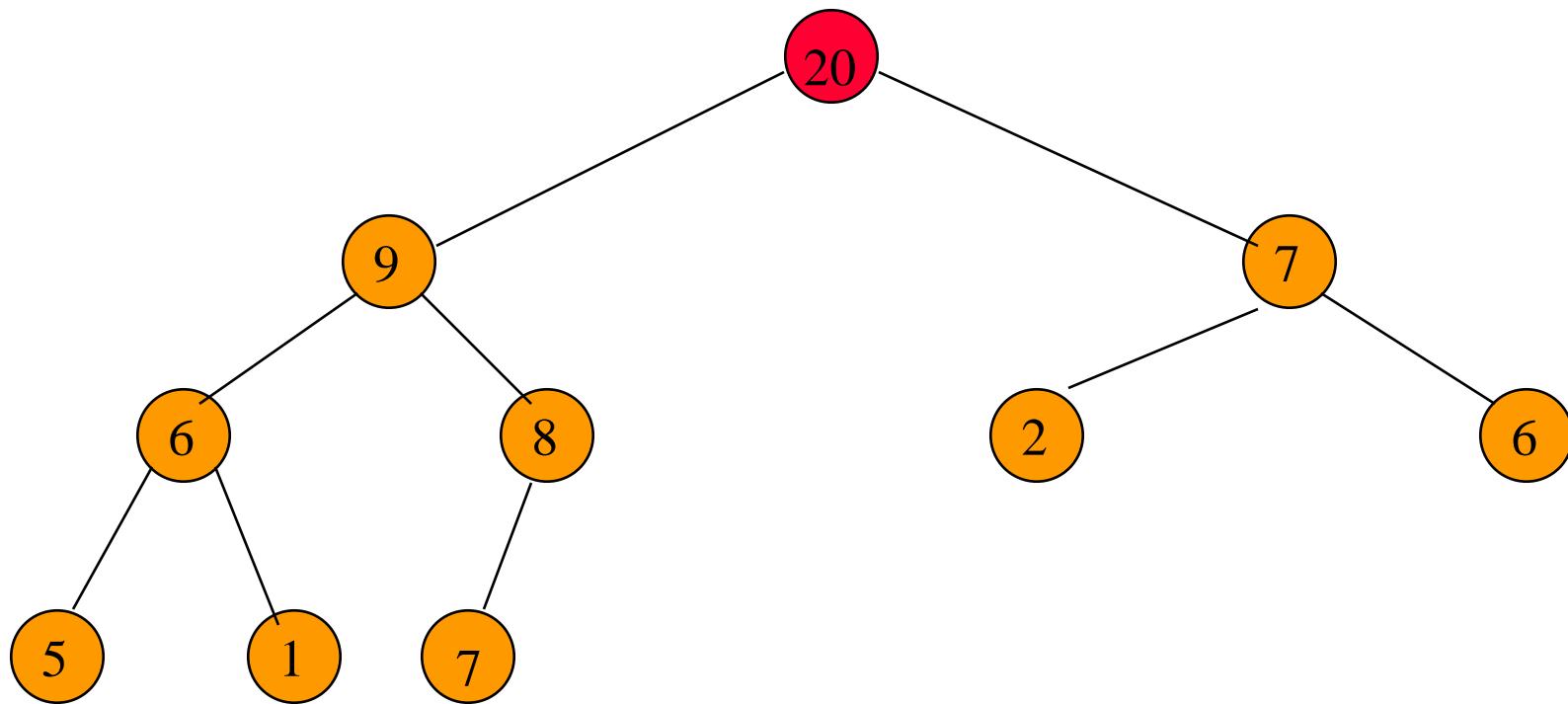
New element is 20.

Putting An Element Into A Max Heap



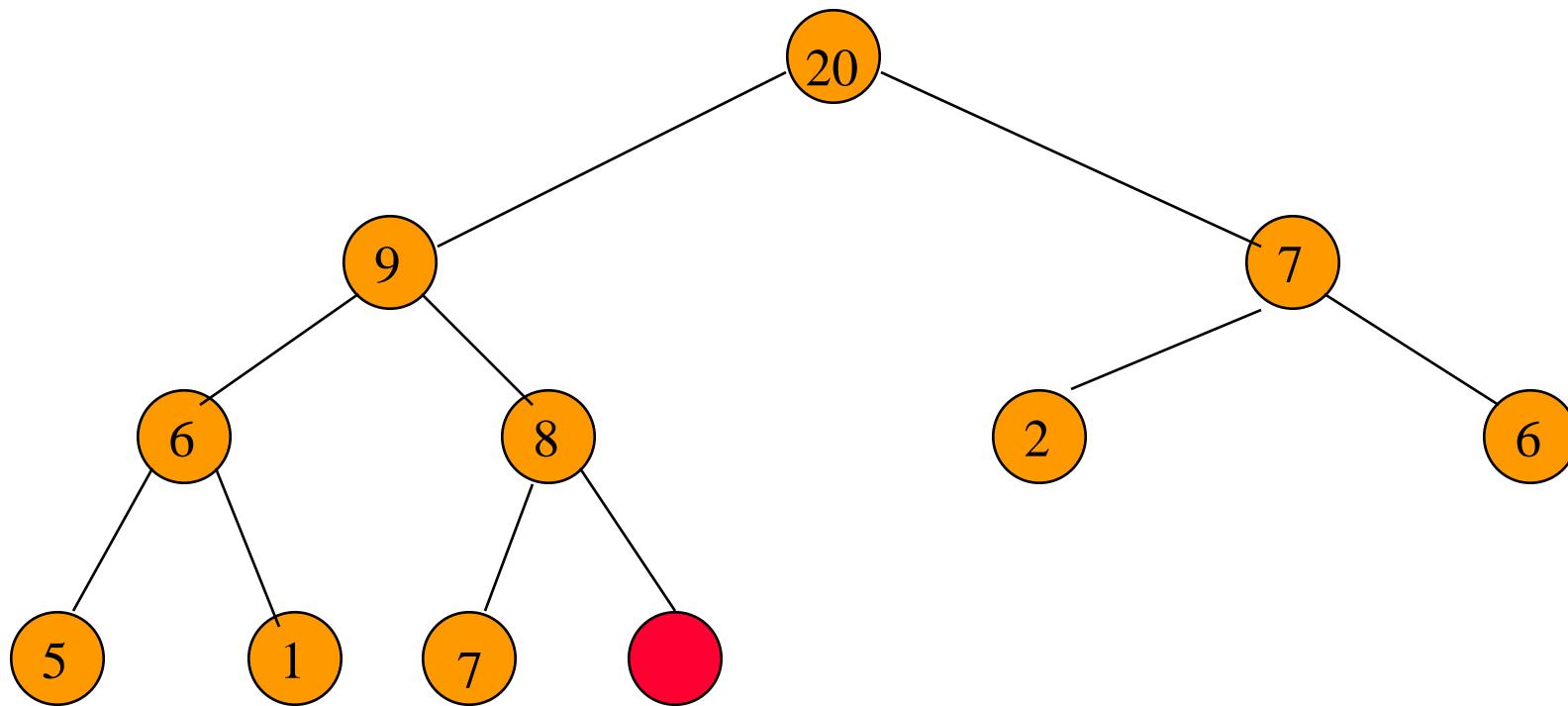
New element is 20.

Putting An Element Into A Max Heap



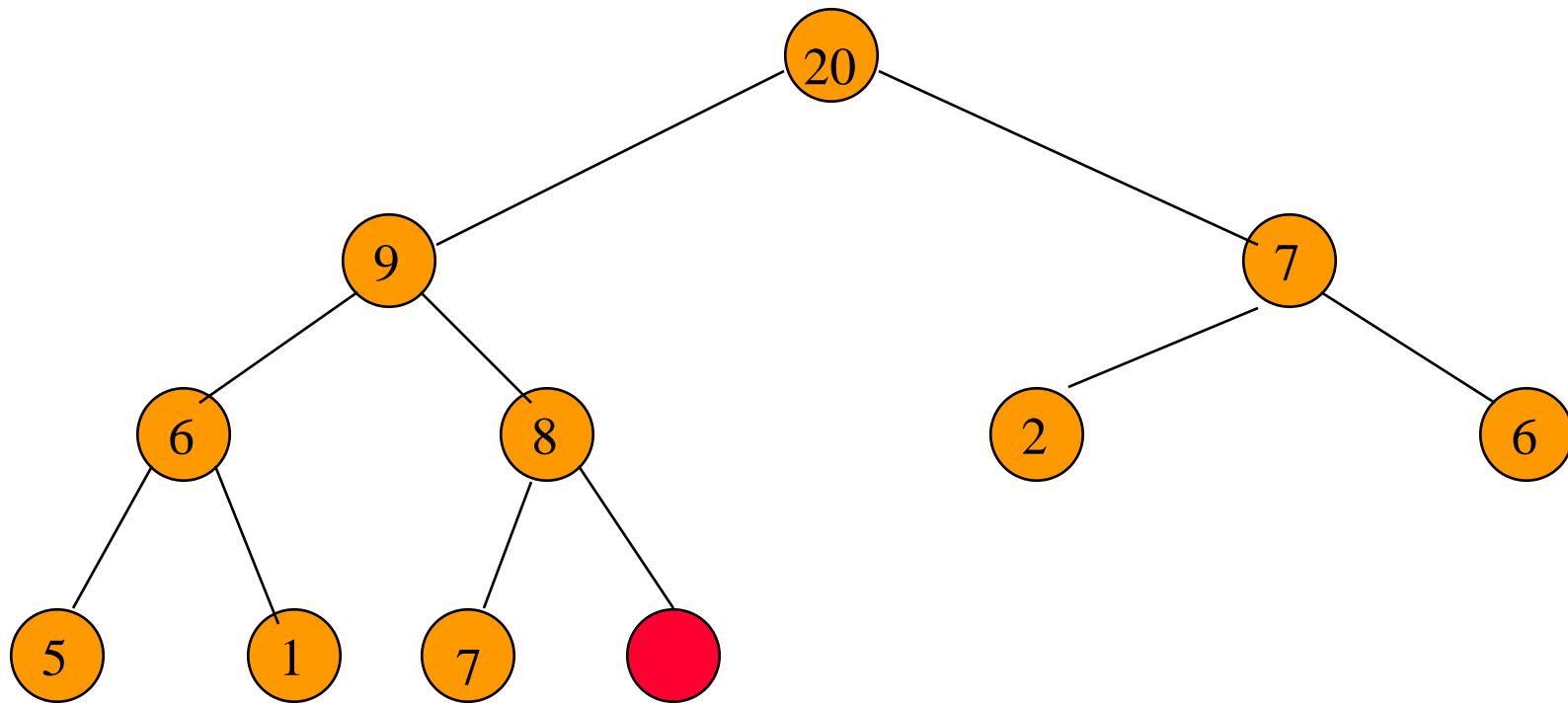
New element is 20.

Putting An Element Into A Max Heap



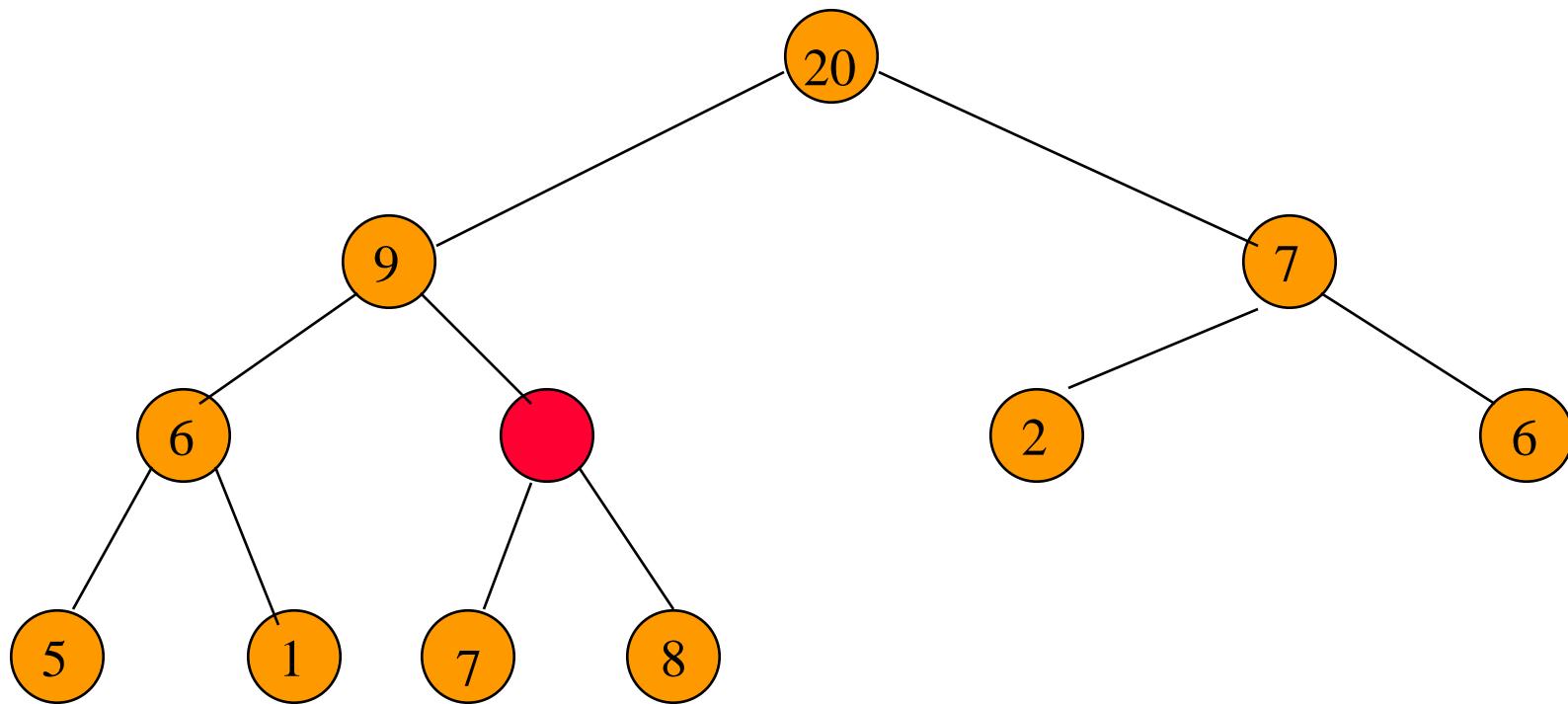
Complete binary tree with 11 nodes.

Putting An Element Into A Max Heap



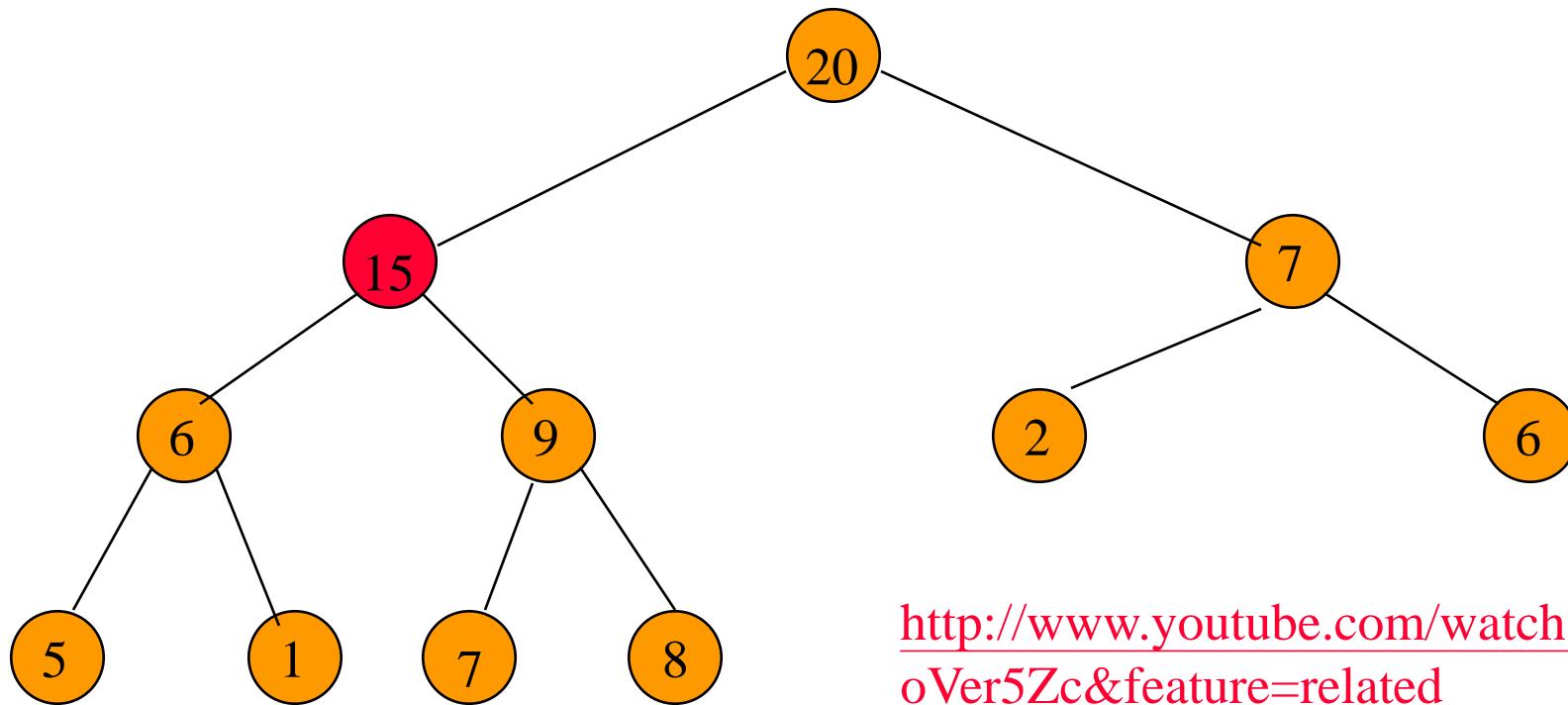
New element is 15.

Putting An Element Into A Max Heap



New element is 15.

Putting An Element Into A Max Heap



<http://www.youtube.com/watch?v=wkWHoVer5Zc&feature=related>

New element is 15.



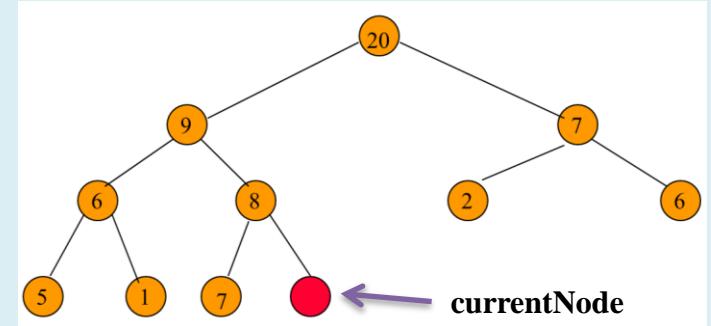
MaxHeap put

```
# Function to put a node into the heap
def put(self, theElement):

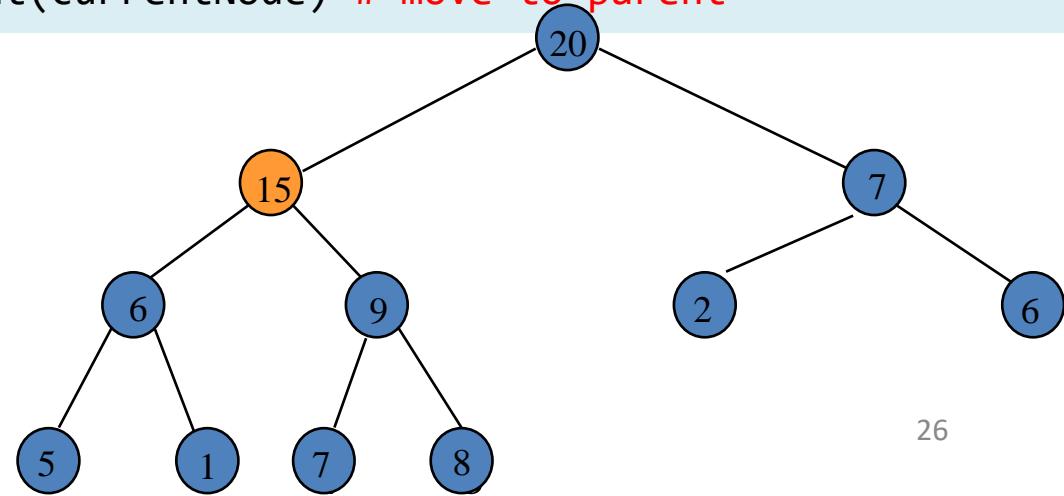
    if self.size >= self.maxsize:
        return
    # find place for theElement
    # currentNode starts at new leaf and moves
    self.size += 1
    self.Heap[self.size] = theElement

    currentNode = self.size

    while (self.Heap[currentNode] > self.Heap[self.parent(currentNode)]):
        # cannot put theElement in Heap[currentNode]
        self.swap(currentNode, self.parent(currentNode)) # move element down
        currentNode = self.parent(currentNode) # move to parent
```



demo: MaxHeap.py



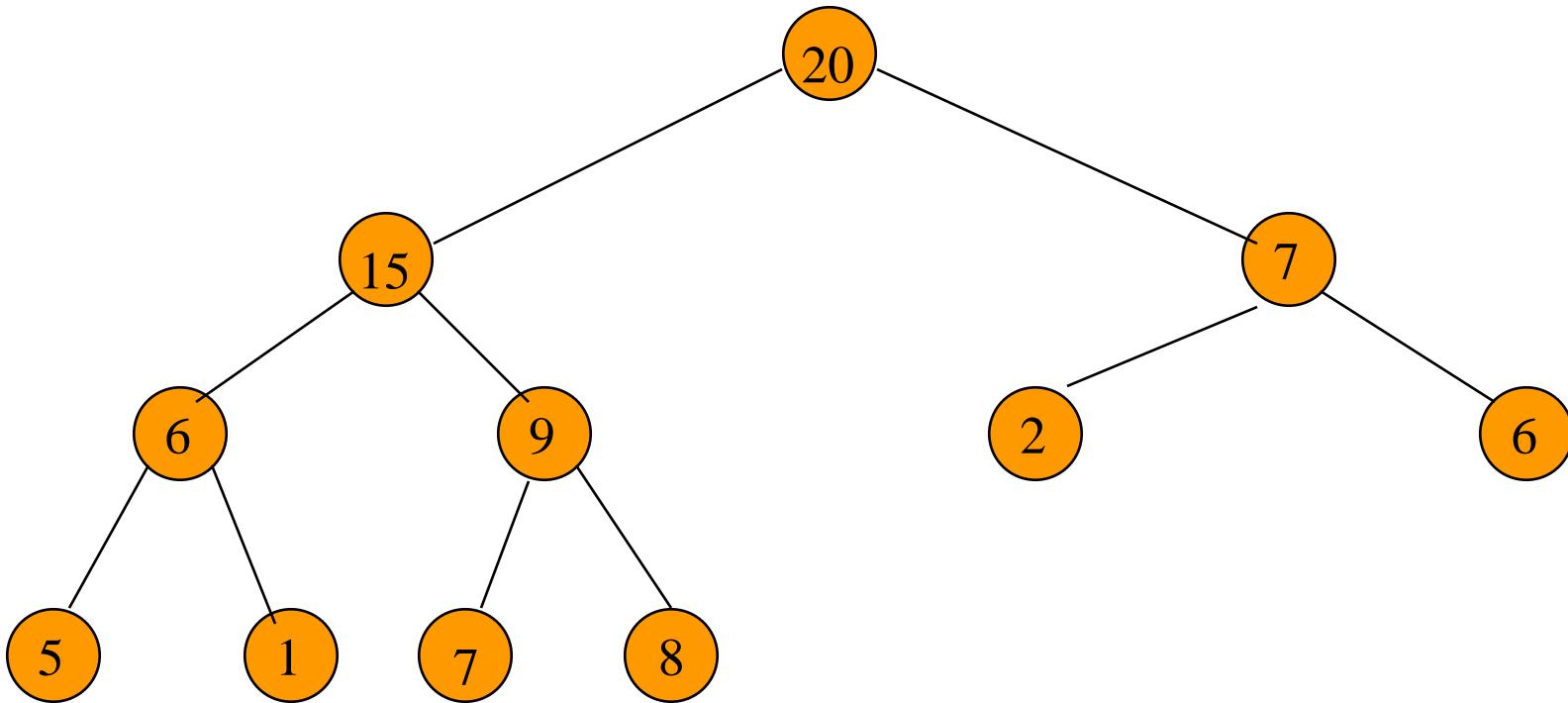
Exercise

- Show the heap (tree) you will have after inserting the following values:
- 80, 40, 30, 60, 81, 90, 100, 10



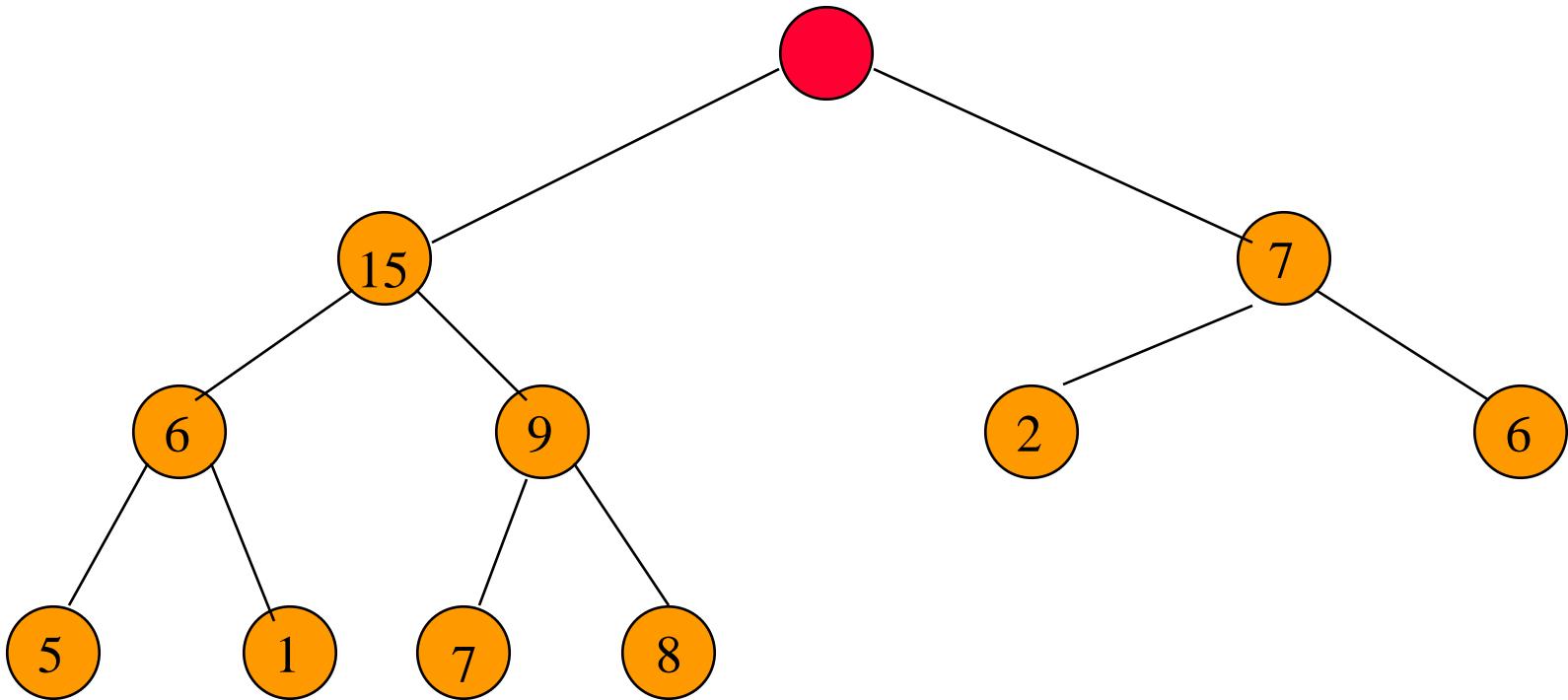
Implementation of Heap - Remove

Removing The Max Element



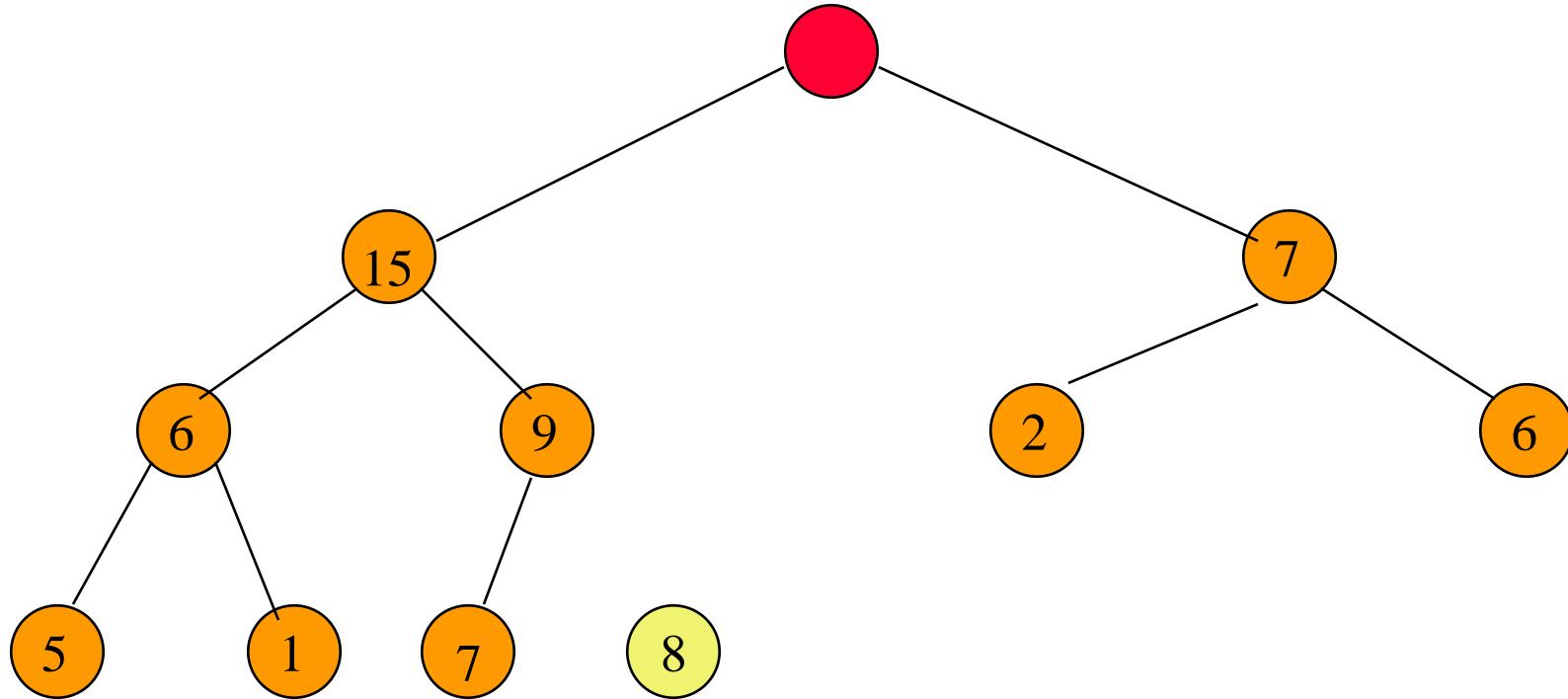
Max element is in the root.

Removing The Max Element



After max element is removed.

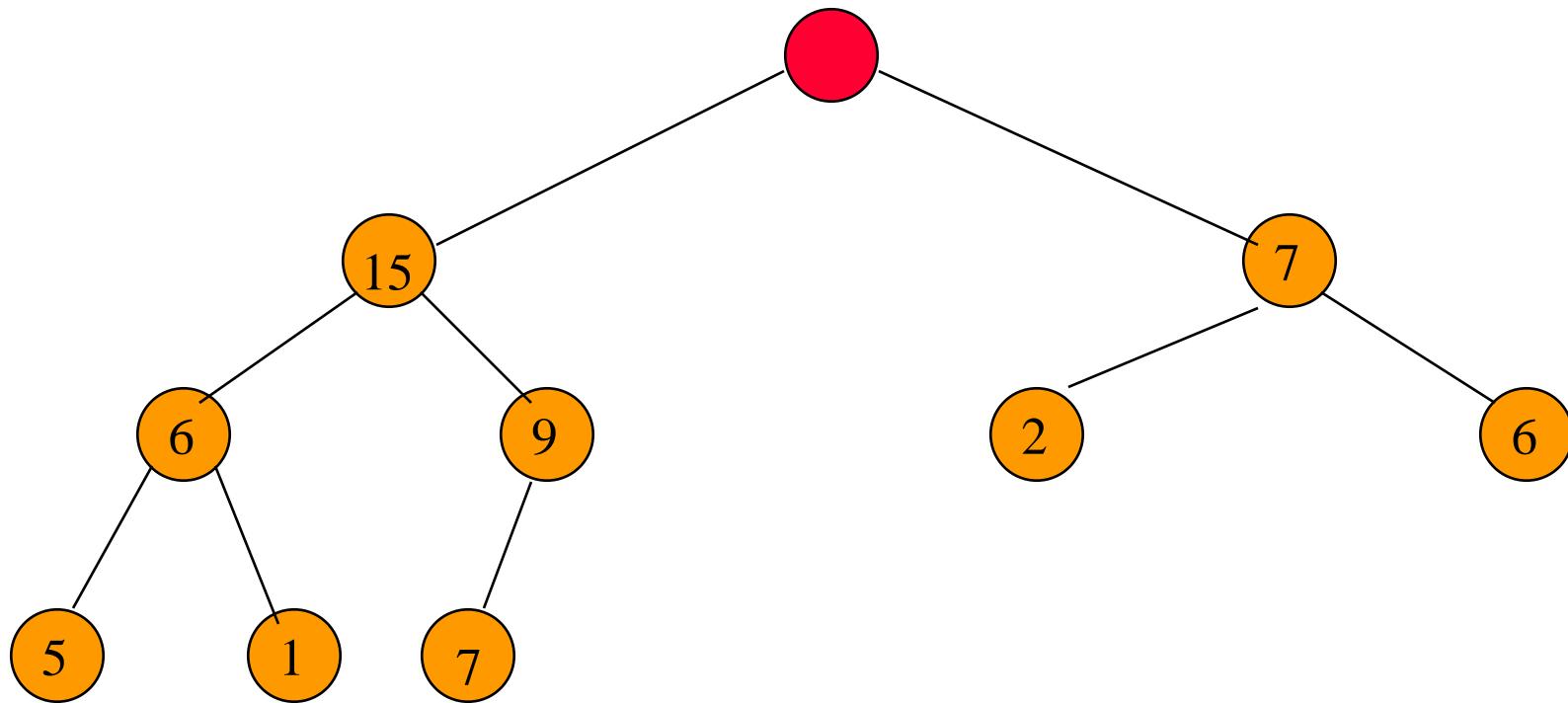
Removing The Max Element



Heap with 10 nodes.

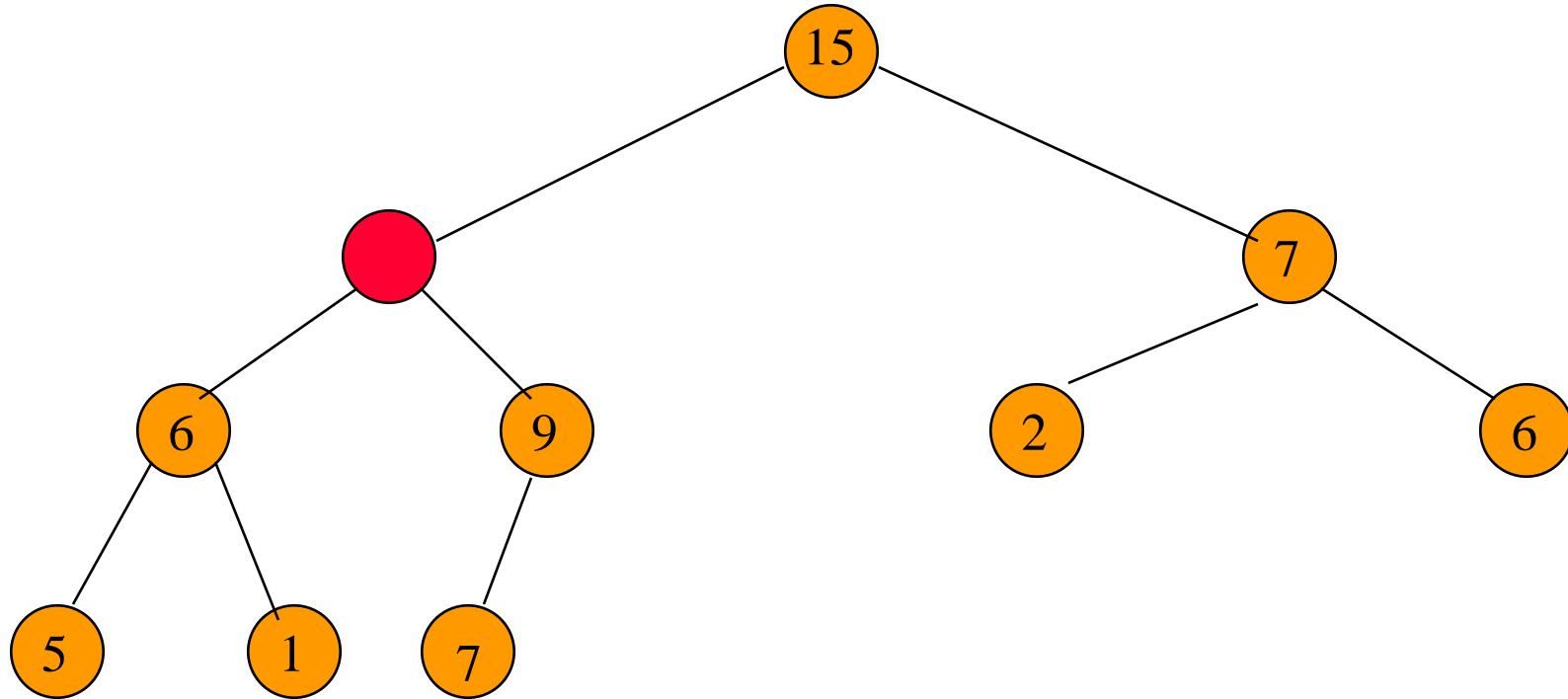
Reinsert 8 into the heap. (at the root)

Removing The Max Element



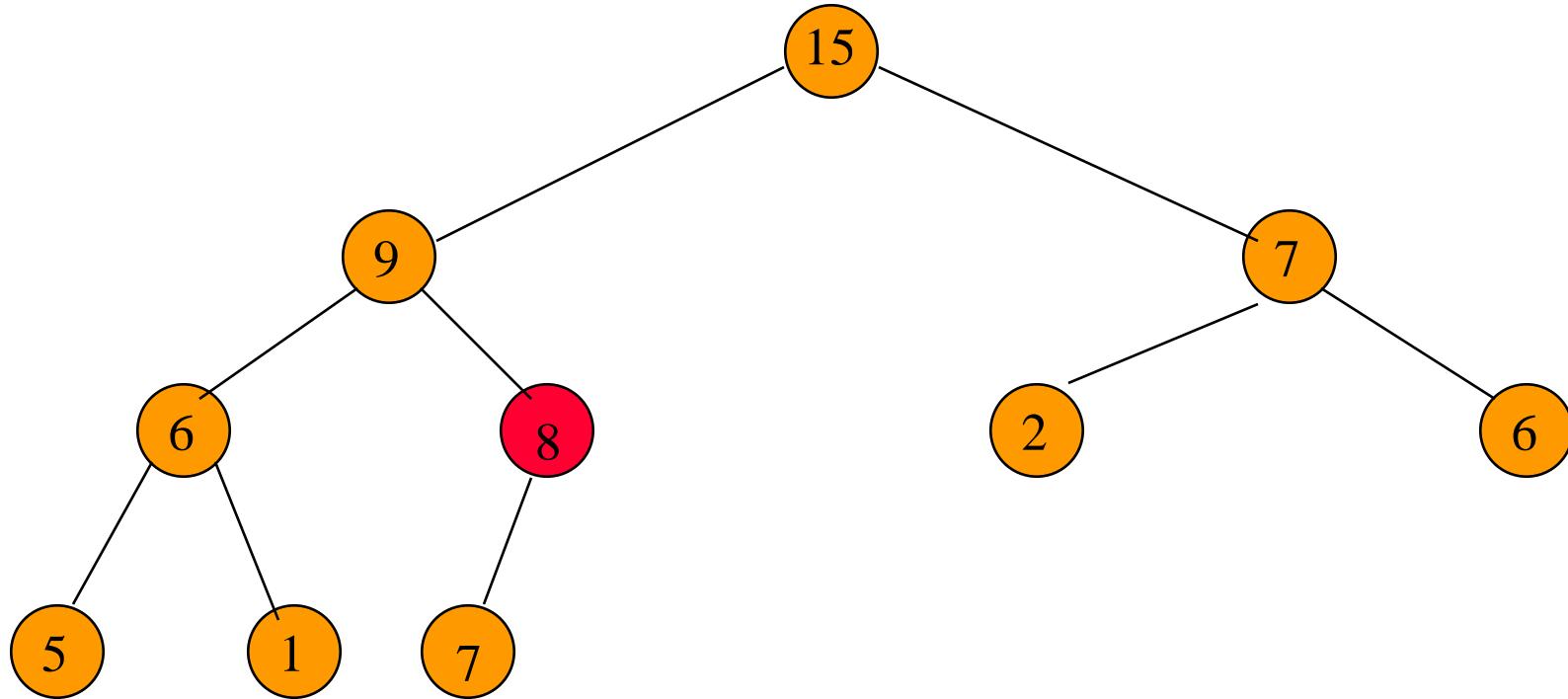
Reinsert 8 into the heap.

Removing The Max Element



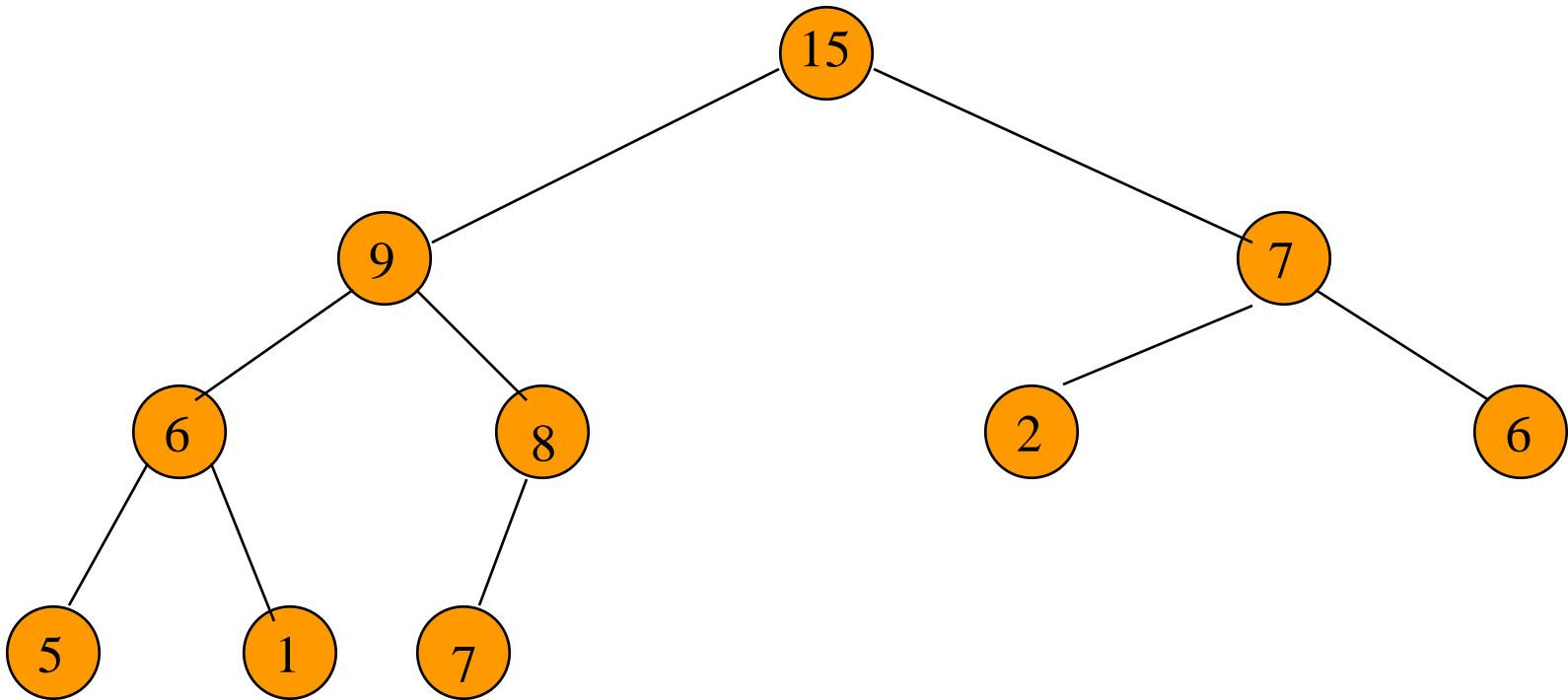
Reinsert 8 into the heap.

Removing The Max Element



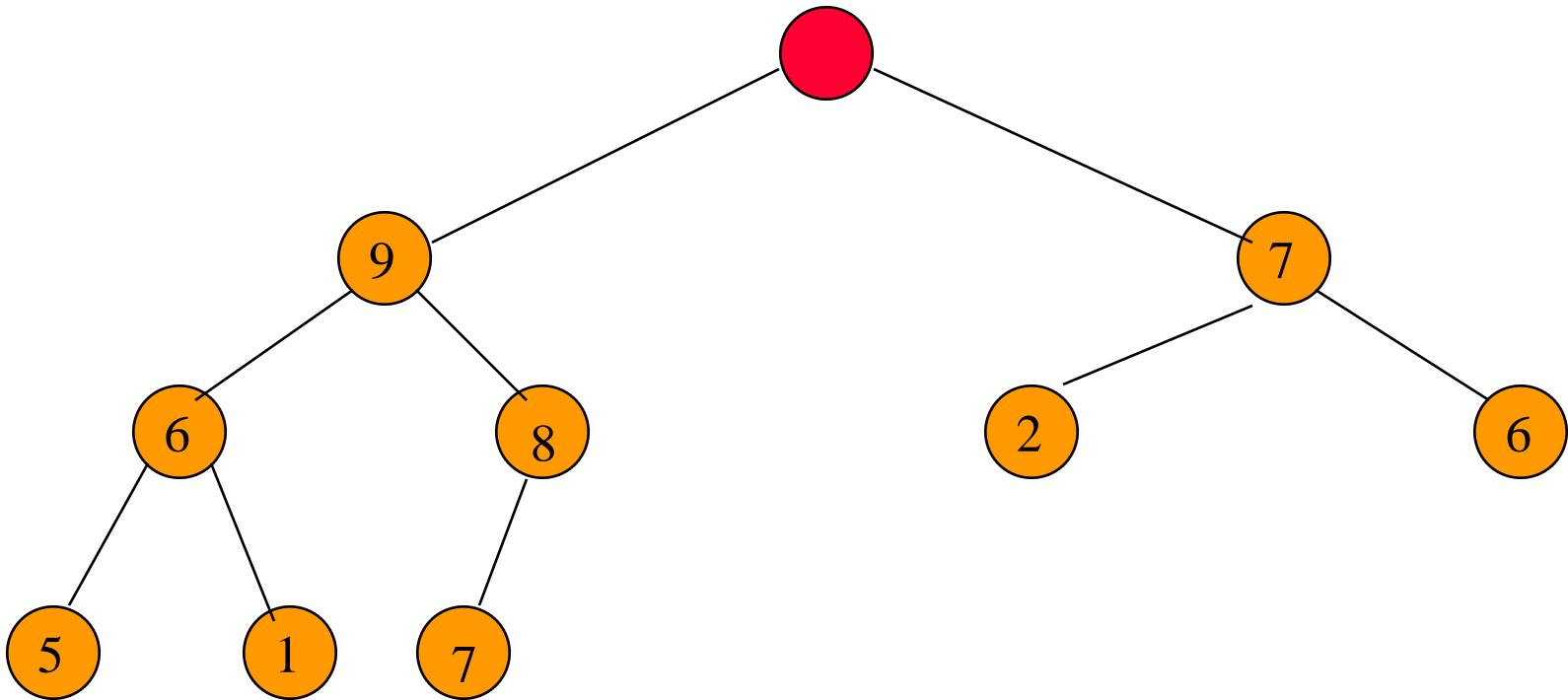
Reinsert 8 into the heap.

Removing The Max Element



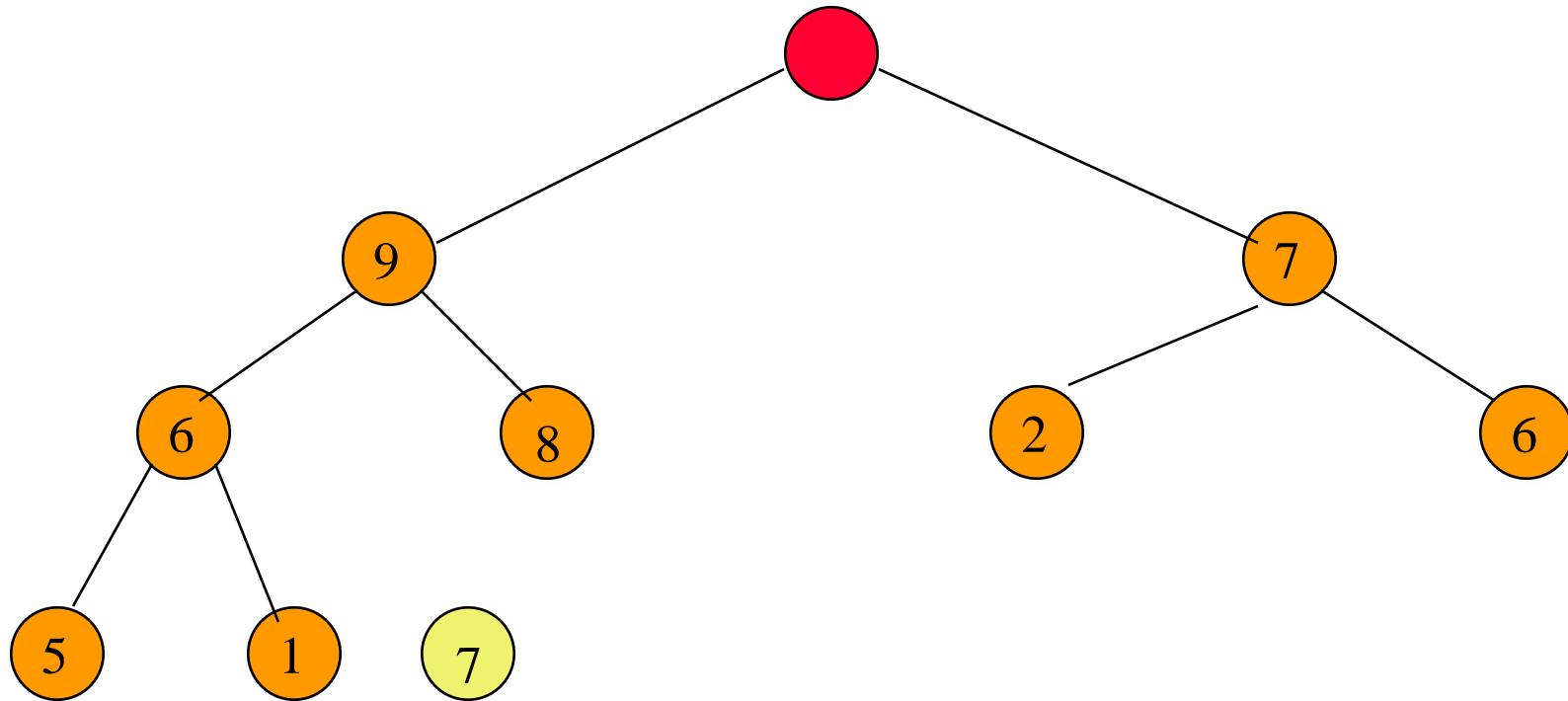
Max element is 15.

Removing The Max Element



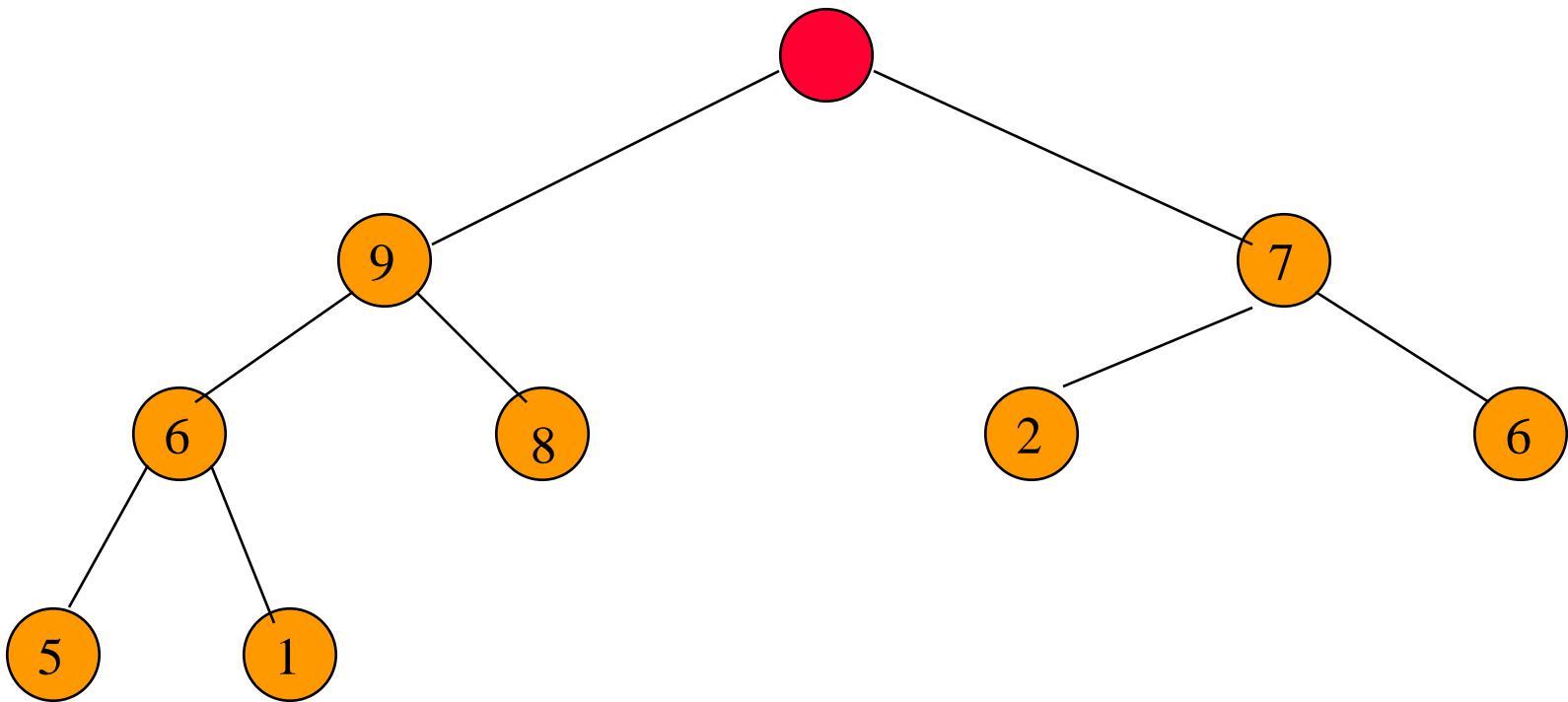
After max element is removed.

Removing The Max Element



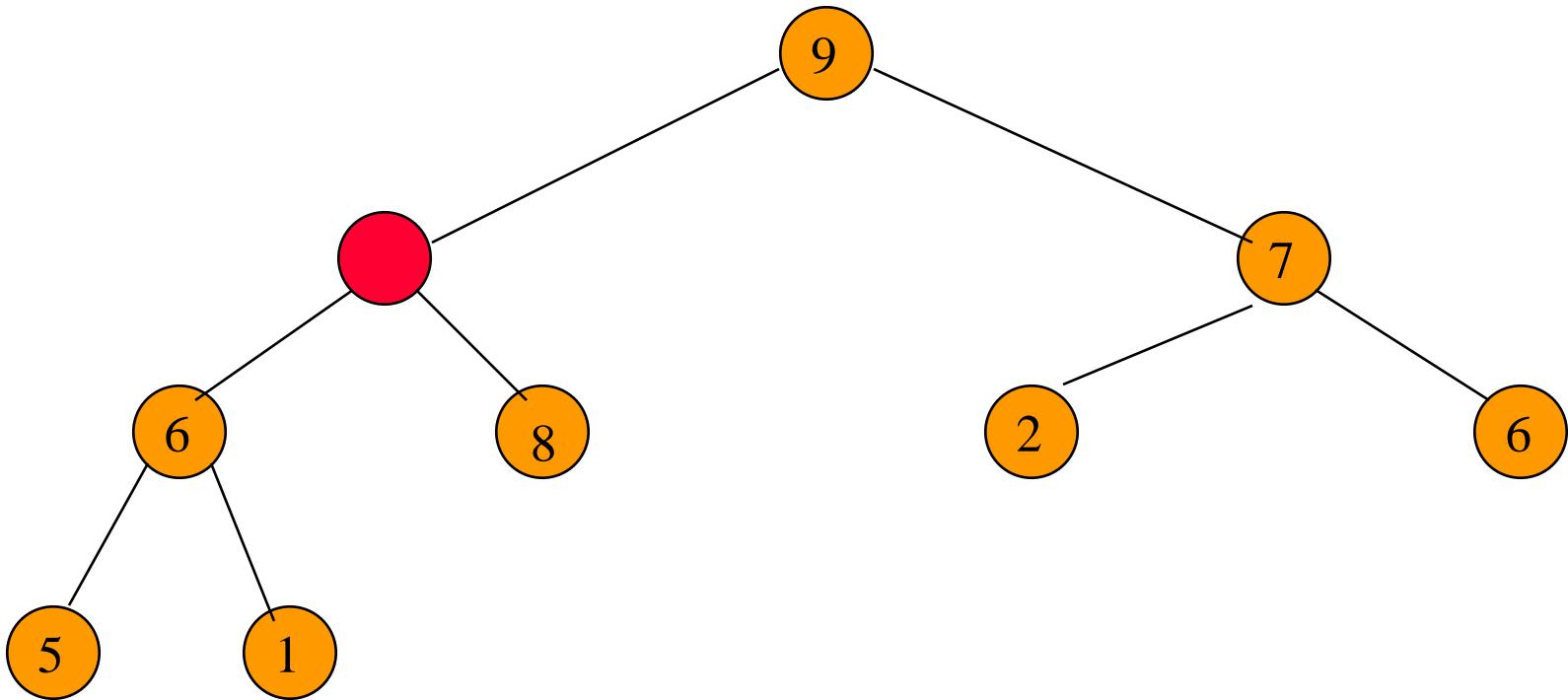
Heap with 9 nodes.

Removing The Max Element



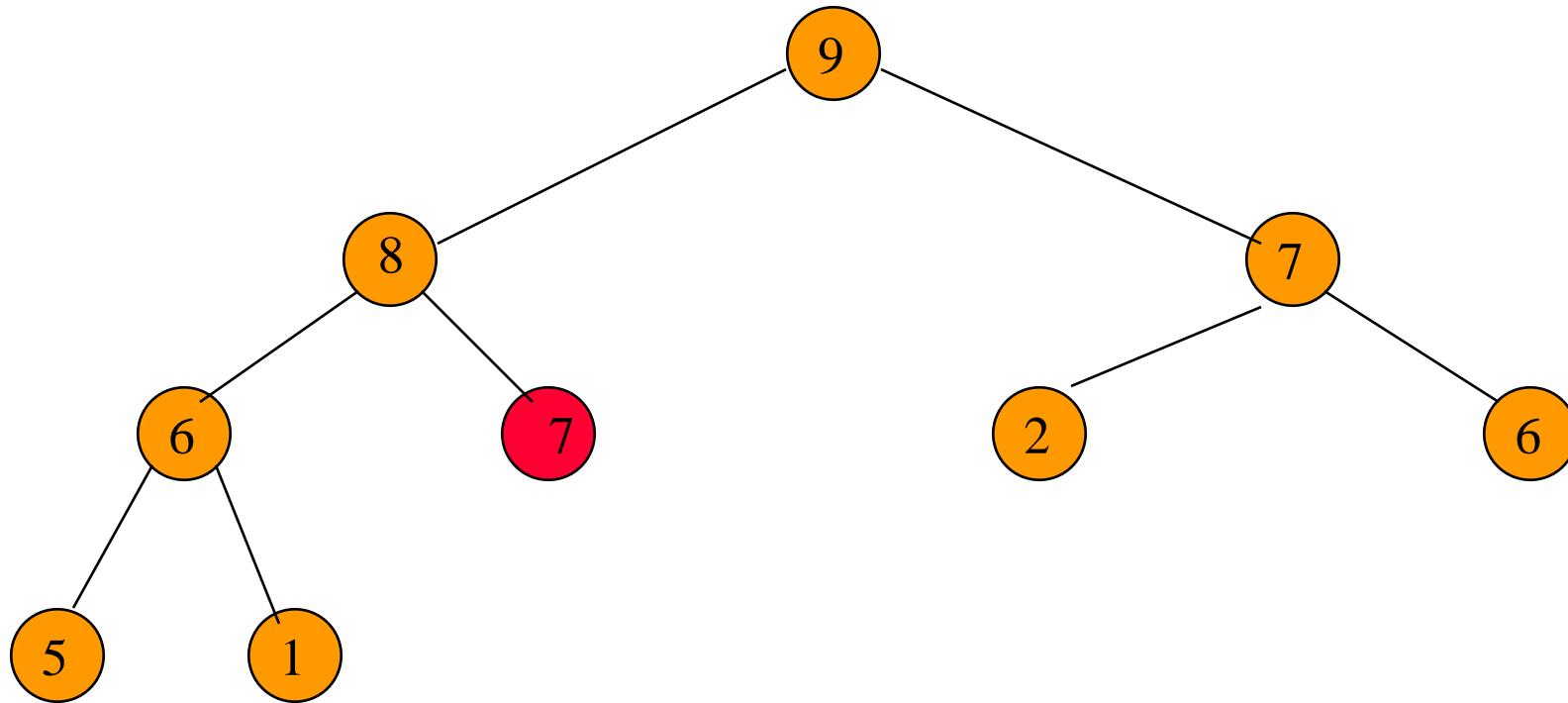
Reinsert 7.

Removing The Max Element



Reinsert 7.

Removing The Max Element

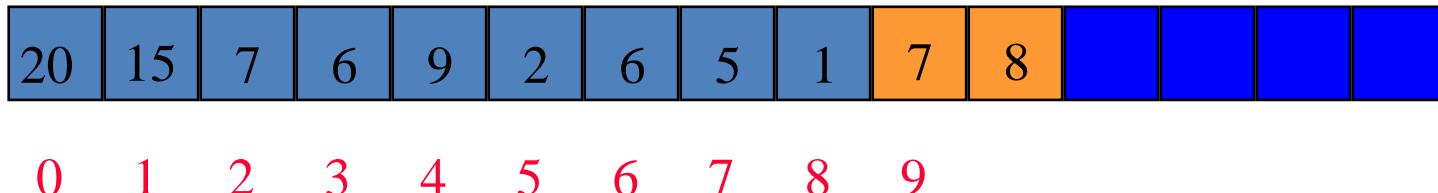


Reinsert 7.

MaxHeap remove

```
# Function to remove and return the maximum element from the
# heap
def removeMax(self):

    popped = self.Heap[self.FRONT] # max element
    # reheapify
    self.Heap[self.FRONT] = self.Heap[self.size]
    self.size -= 1 # decrease the size
    self.maxHeapify(self.FRONT) ————— heapify() method is shown in
                                the coming slides
    return popped
```

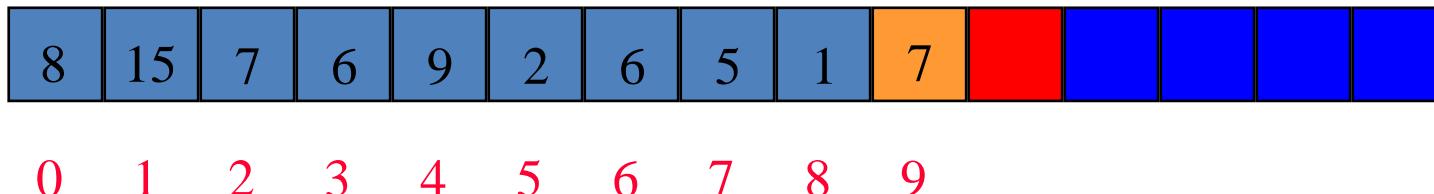
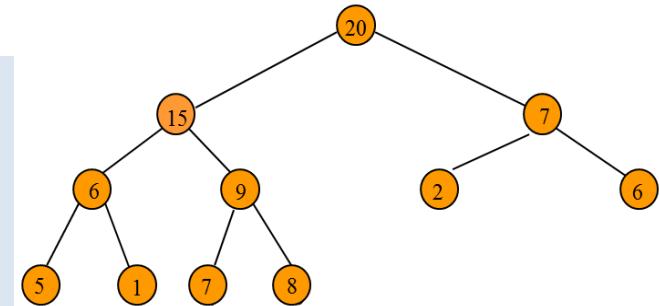


MaxHeap heapify

```
# Function to heapify the node at pos
def maxHeapify(self, pos):
    # If the node is a non-leaf node and smaller
    # than any of its child
    if not self.isLeaf(pos):
        if (self.Heap[pos] < self.Heap[self.leftChild(pos)] or
            self.Heap[pos] < self.Heap[self.rightChild(pos)]):

            # Swap with the left child and heapify the left child
            if (self.Heap[self.leftChild(pos)] >
                self.Heap[self.rightChild(pos)]):
                self.swap(pos, self.leftChild(pos))
                self.maxHeapify(self.leftChild(pos))

            # Swap with the right child and heapify the right child
        else:
            self.swap(pos, self.rightChild(pos))
            self.maxHeapify(self.rightChild(pos))
```



MaxHeap

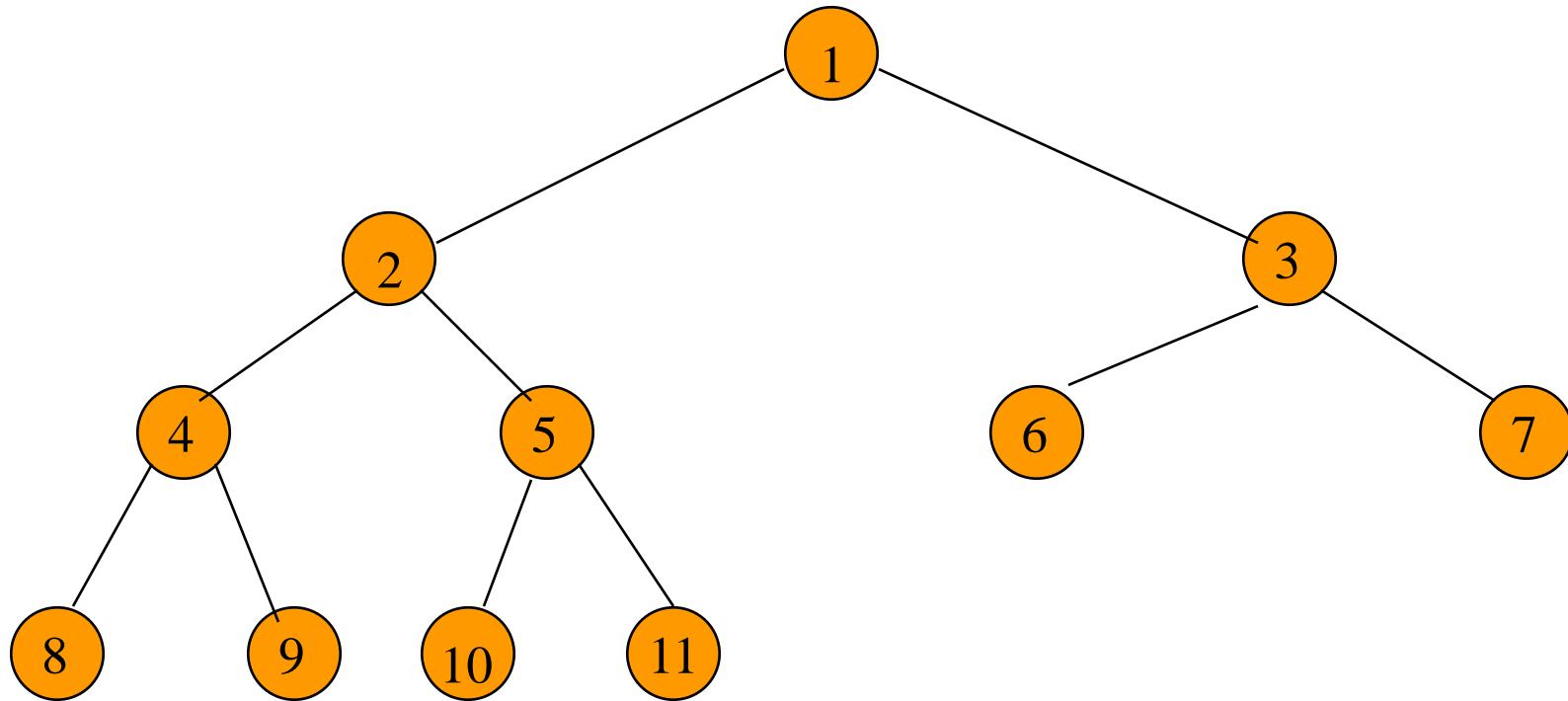
to simplify the implementation, index start from 1

```
def __init__(self, maxsize):  
    self.maxsize = maxsize  
    self.size = 0  
    self.Heap = [0] * (self.maxsize + 1)  
    self.Heap[0] = sys.maxsize  
    self.FRONT = 1  
    self.printed = False
```

demo: MaxHeap.py

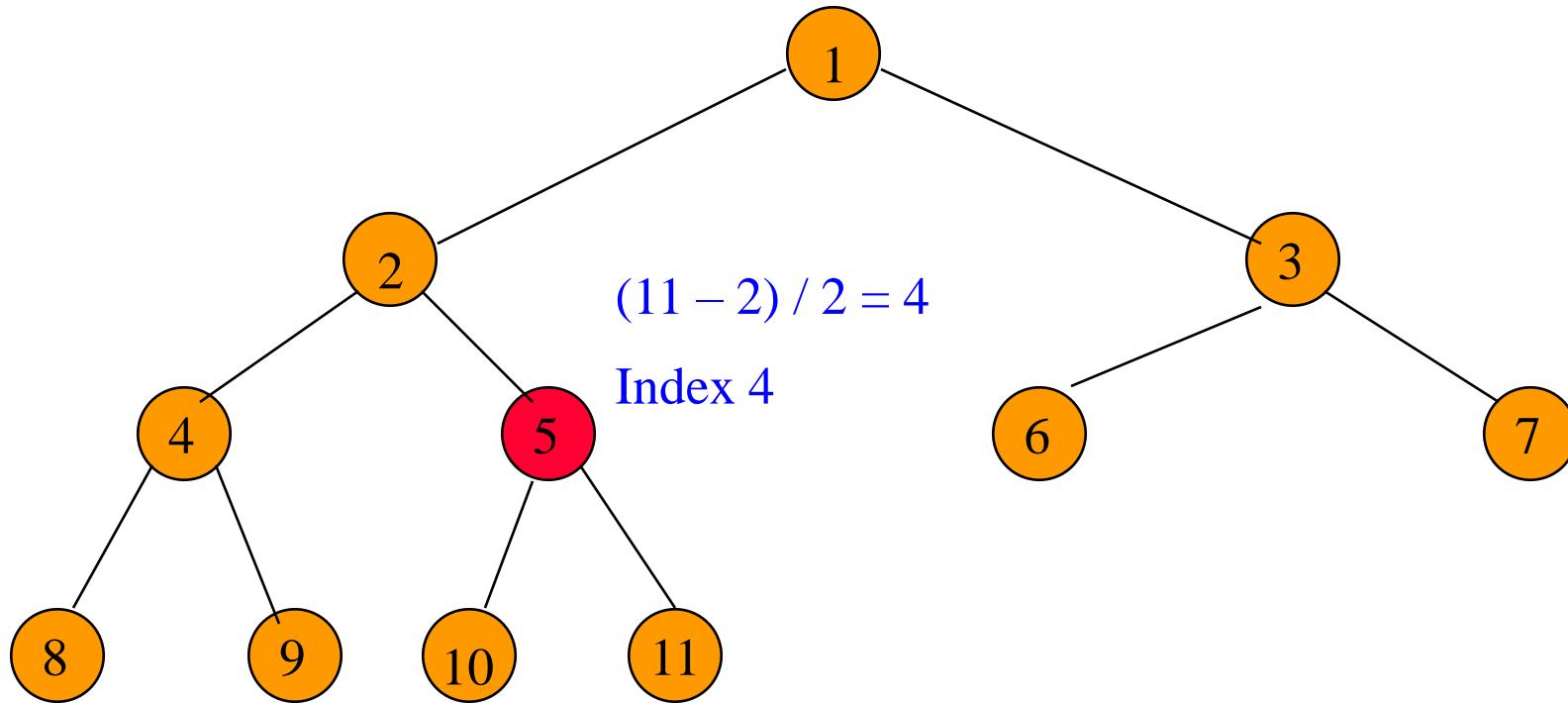
Initializing

Initializing A Max Heap



input array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

Initializing A Max Heap

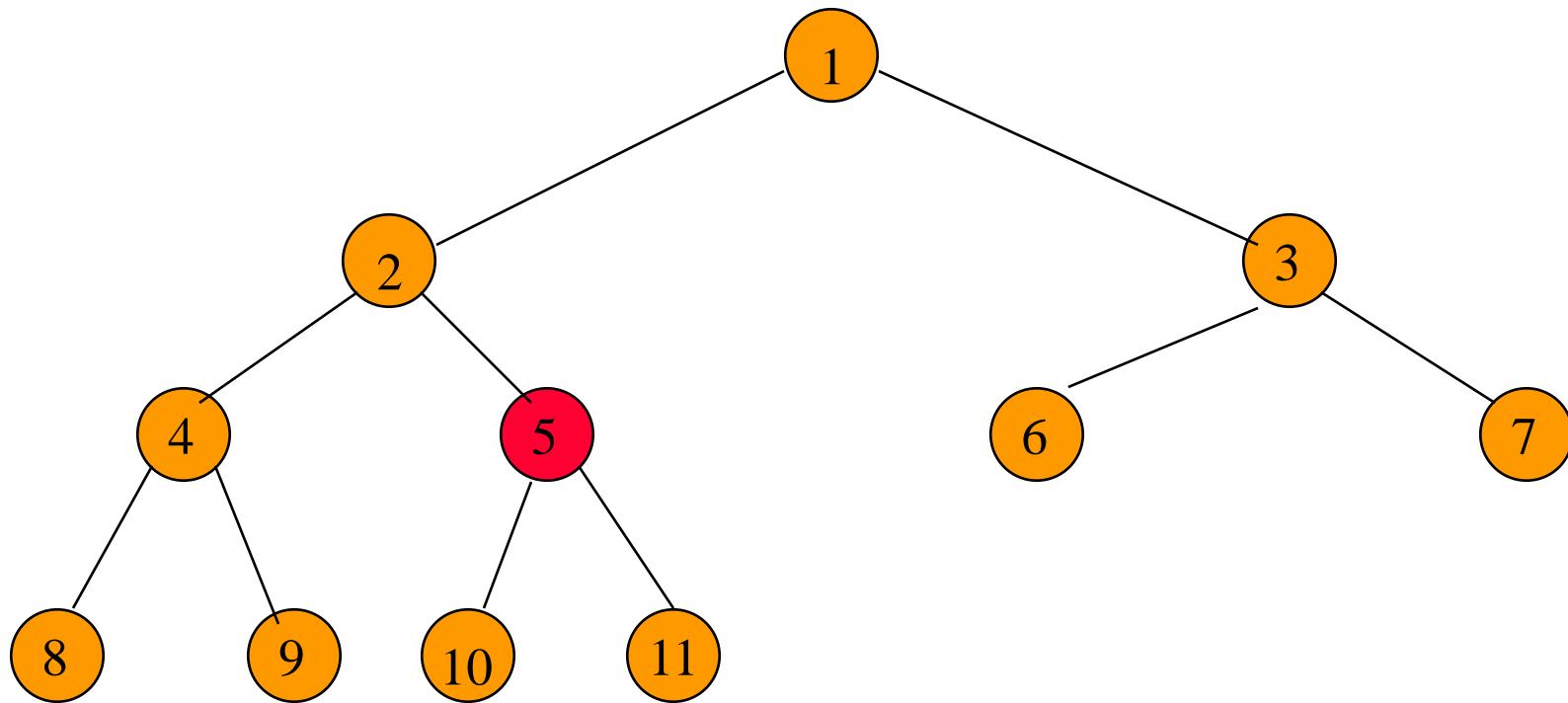


input array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

Start at rightmost array position that has a child.

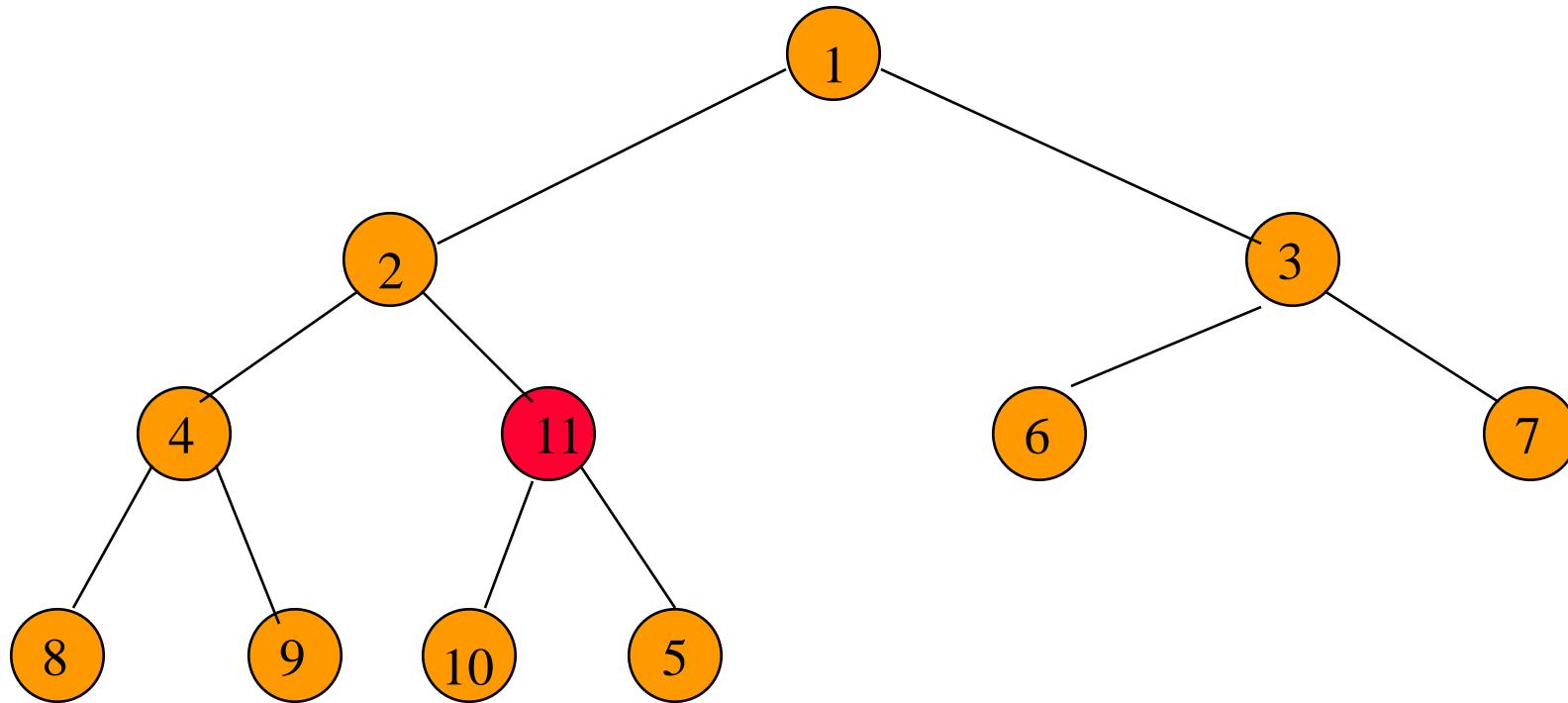
Index is $(n-2)/2$.

Initializing A Max Heap



input array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

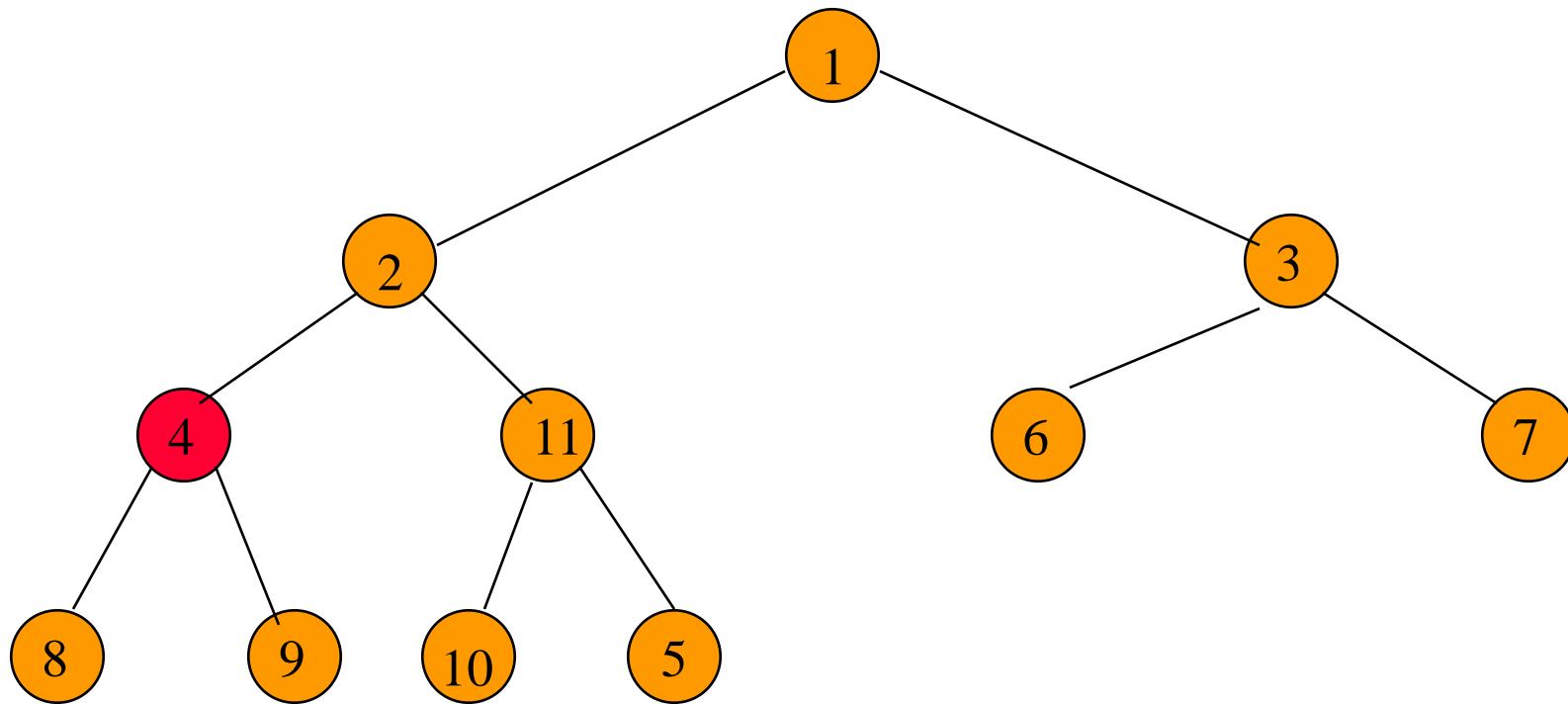
Initializing A Max Heap



input array = [1, 2, 3, 4, 11, 6, 7, 8, 9, 10, 5]

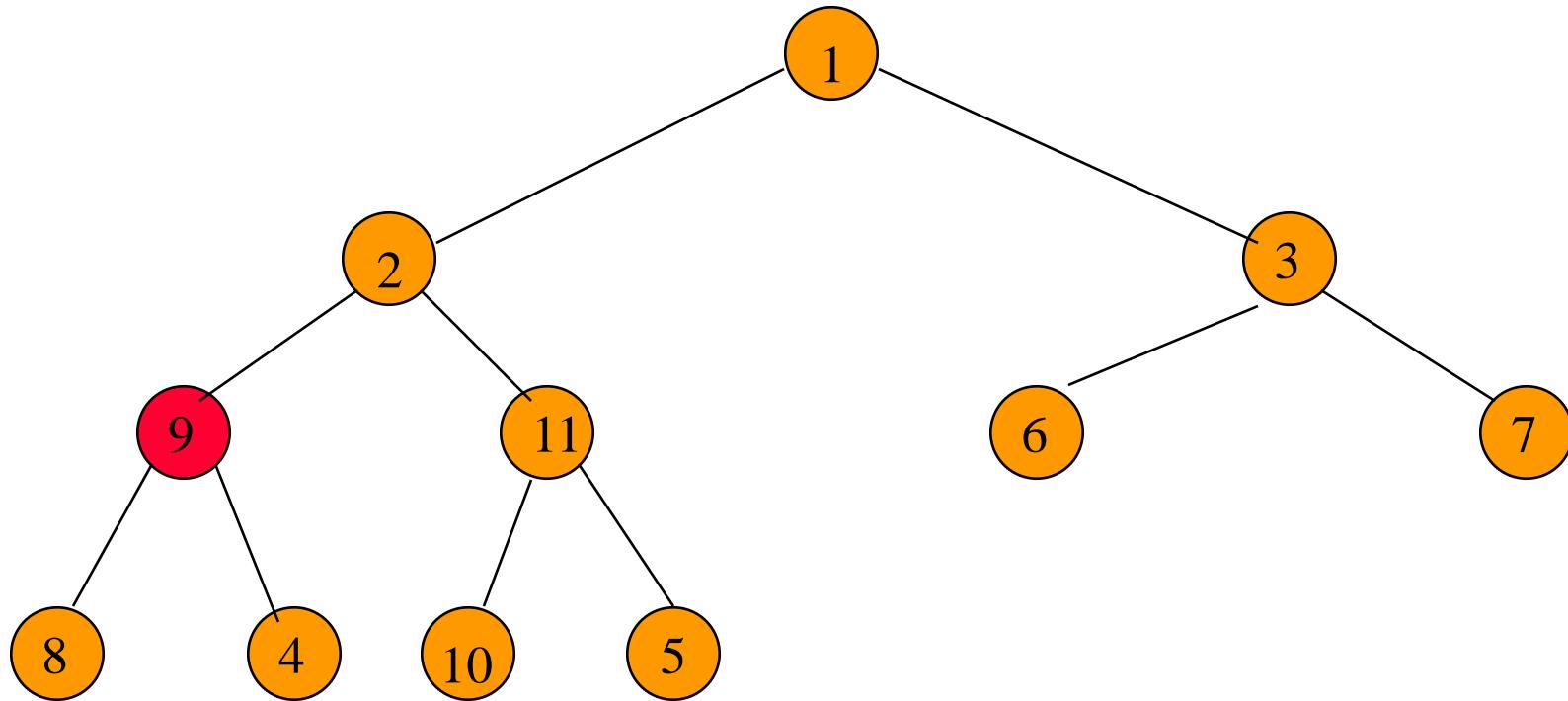
Move to next lower array position.

Initializing A Max Heap



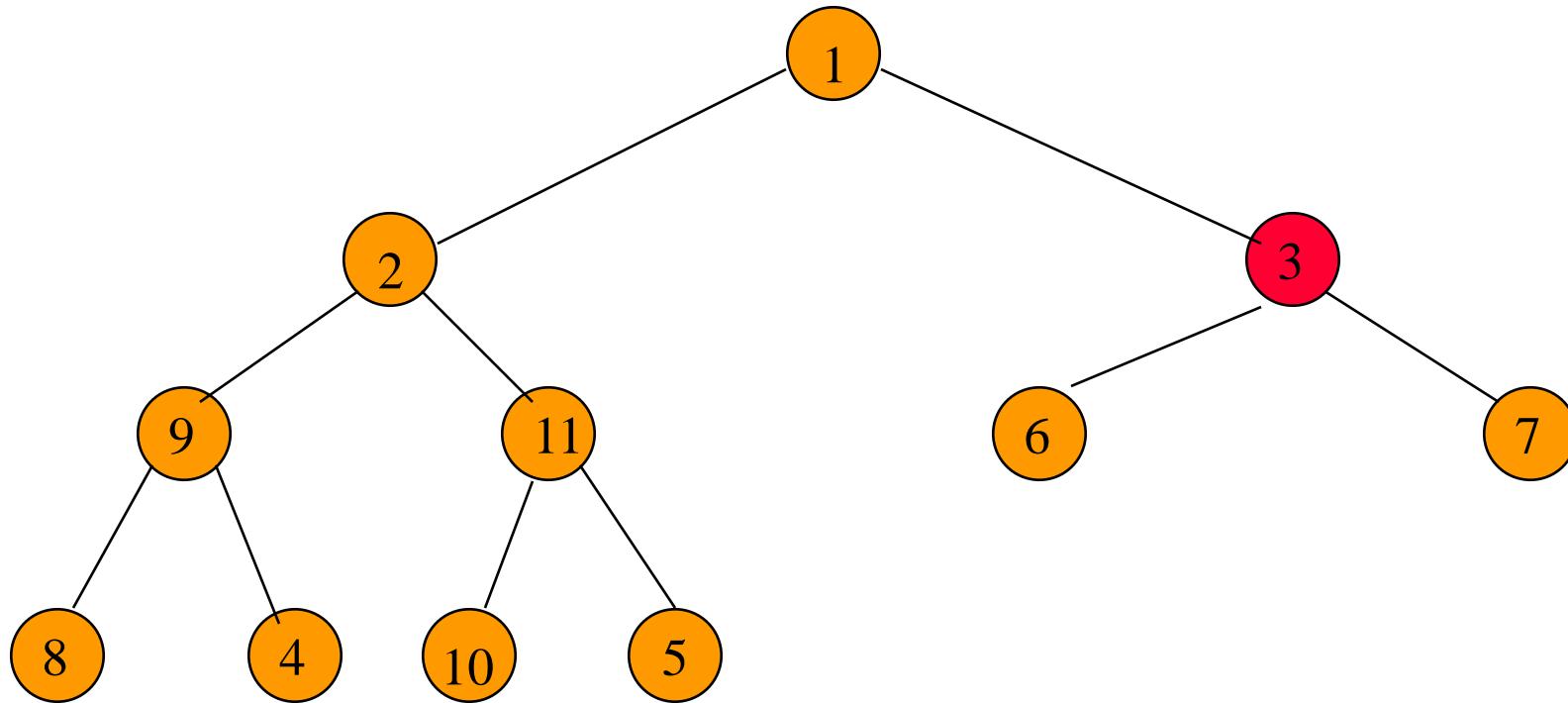
input array = [1, 2, 3, 4, 11, 6, 7, 8, 9, 10, 5]

Initializing A Max Heap



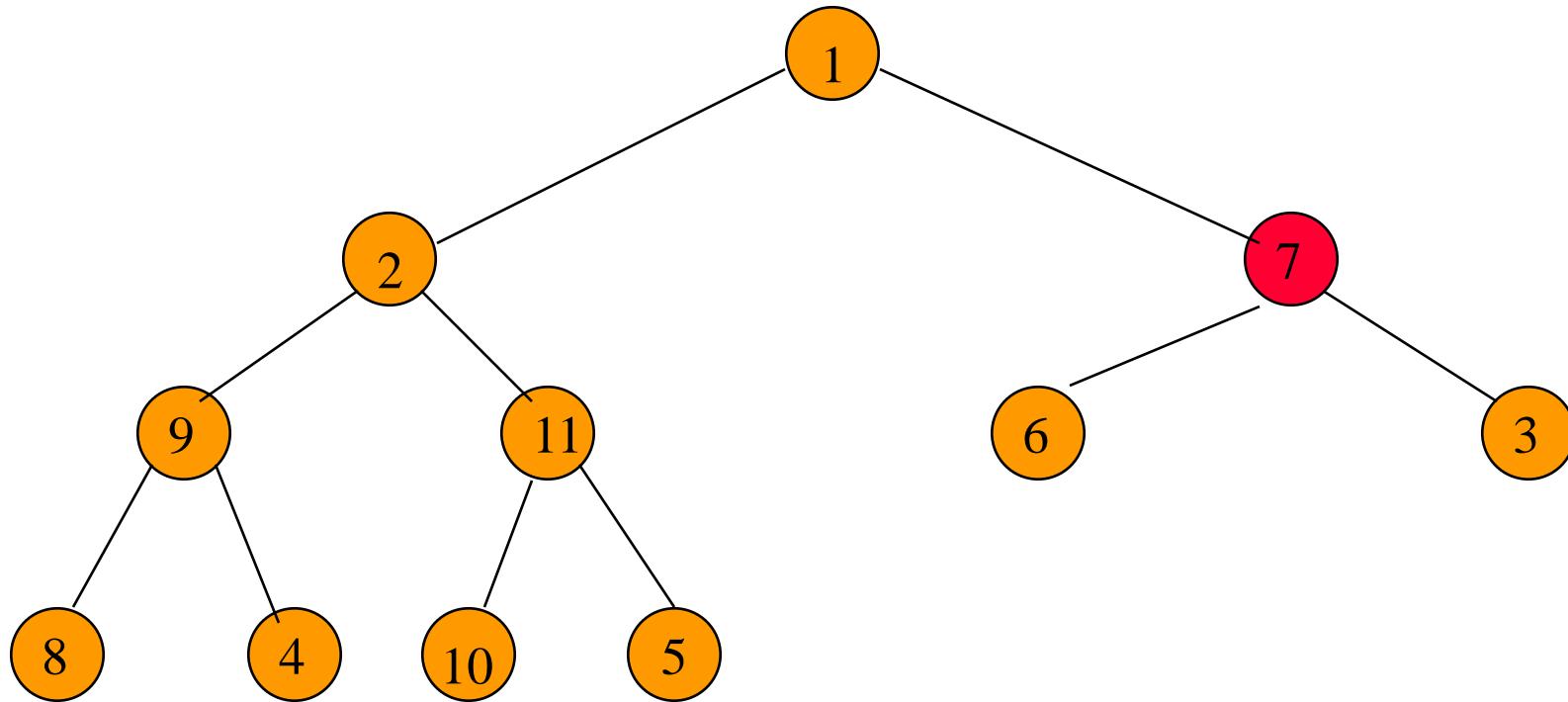
input array = [1, 2, 3, 9, 11, 6, 7, 8, 4, 10, 5]

Initializing A Max Heap



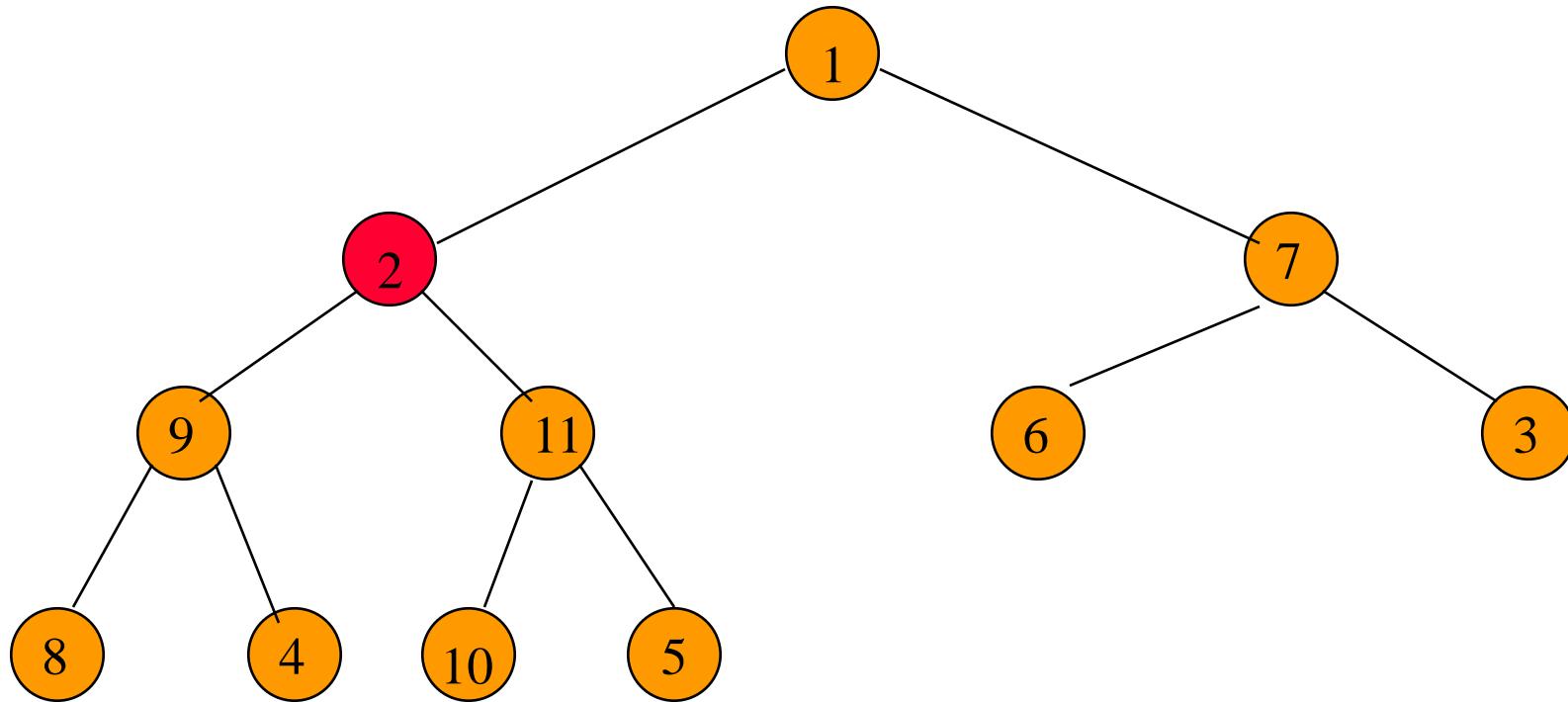
input array = [1, 2, 3, 9, 11, 6, 7, 8, 4, 10, 5]

Initializing A Max Heap



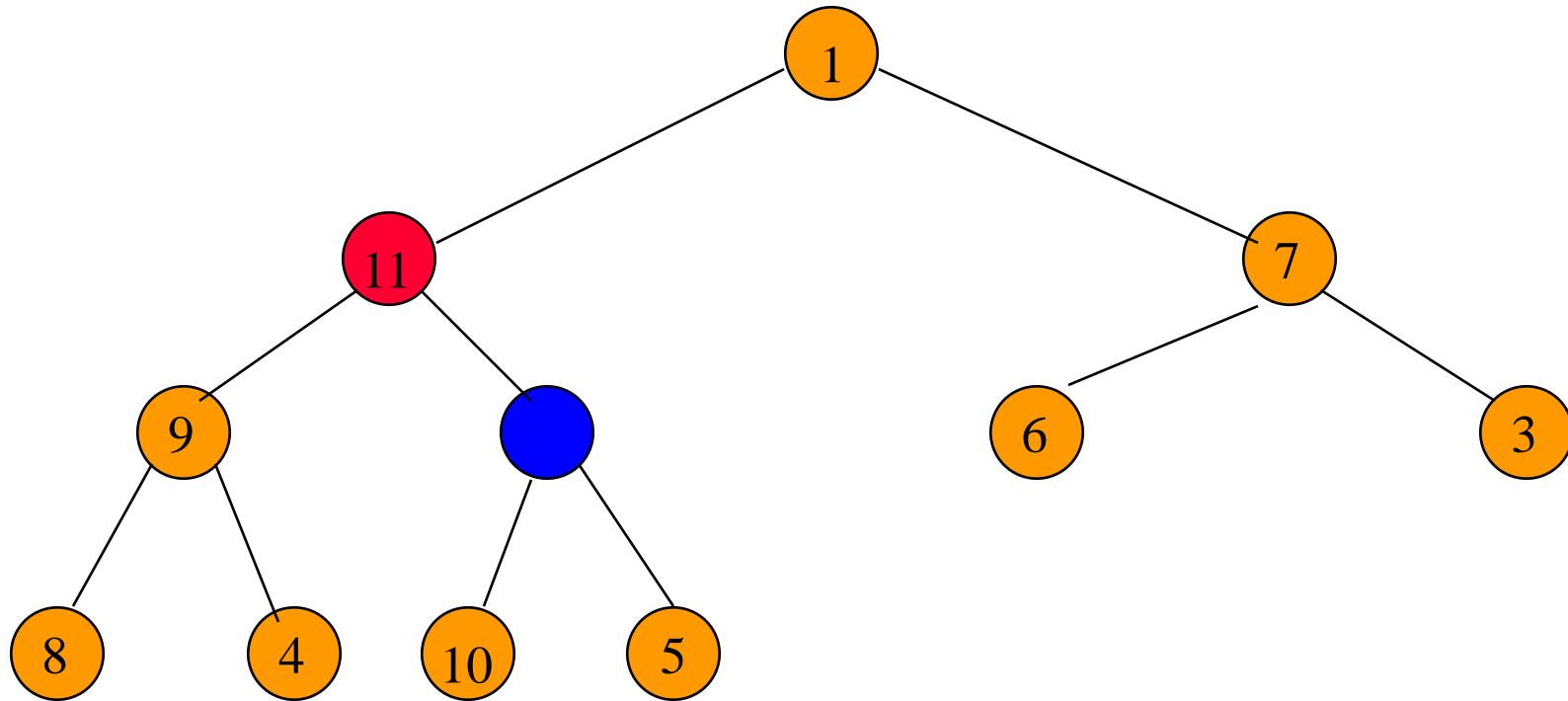
input array = [1, 2, 7, 9, 11, 6, 3, 8, 4, 10, 5]

Initializing A Max Heap



input array = [1, 2, 7, 9, 11, 6, 3, 8, 4, 10, 5]

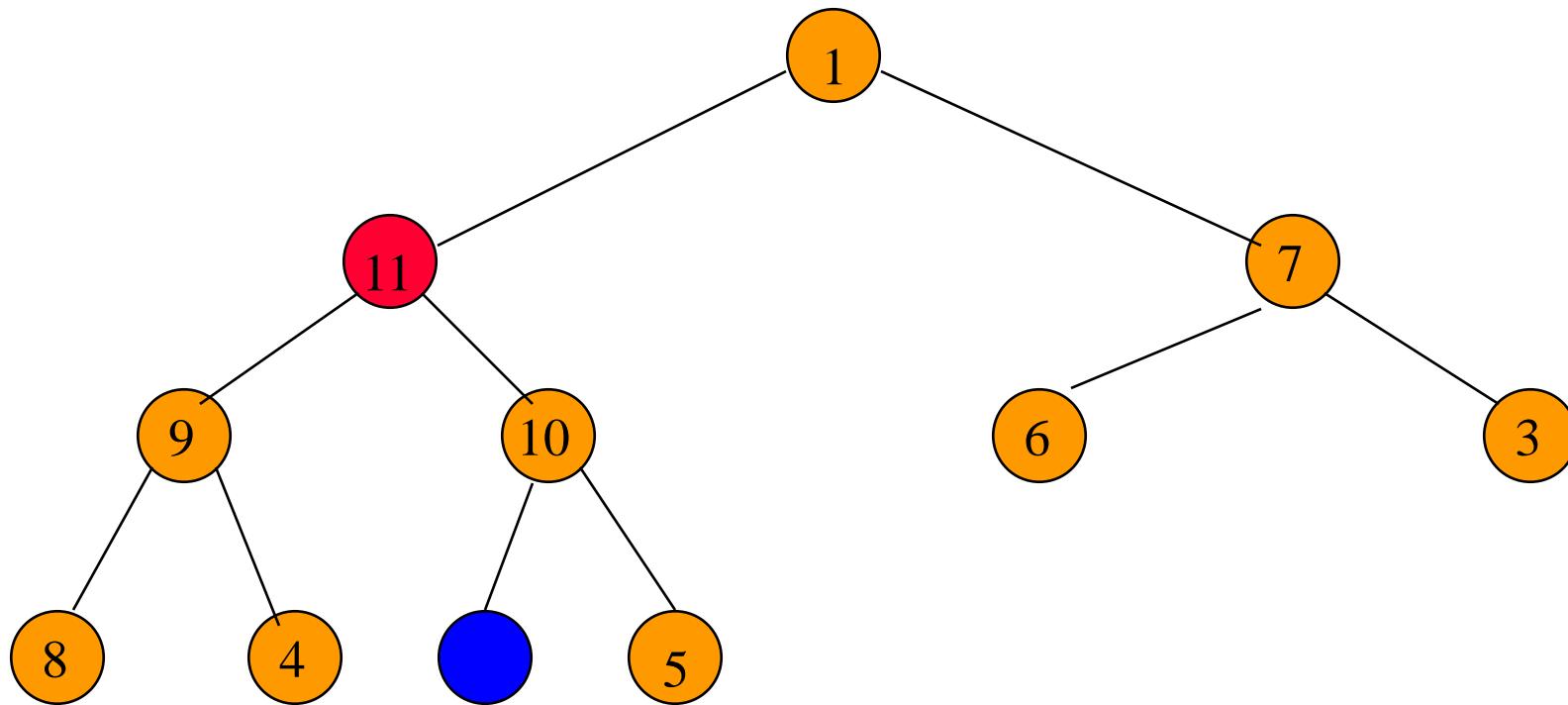
Initializing A Max Heap



input array = [1, 11, 7, 9, 2, 6, 3, 8, 4, 10, 5]

Find a home for 2.

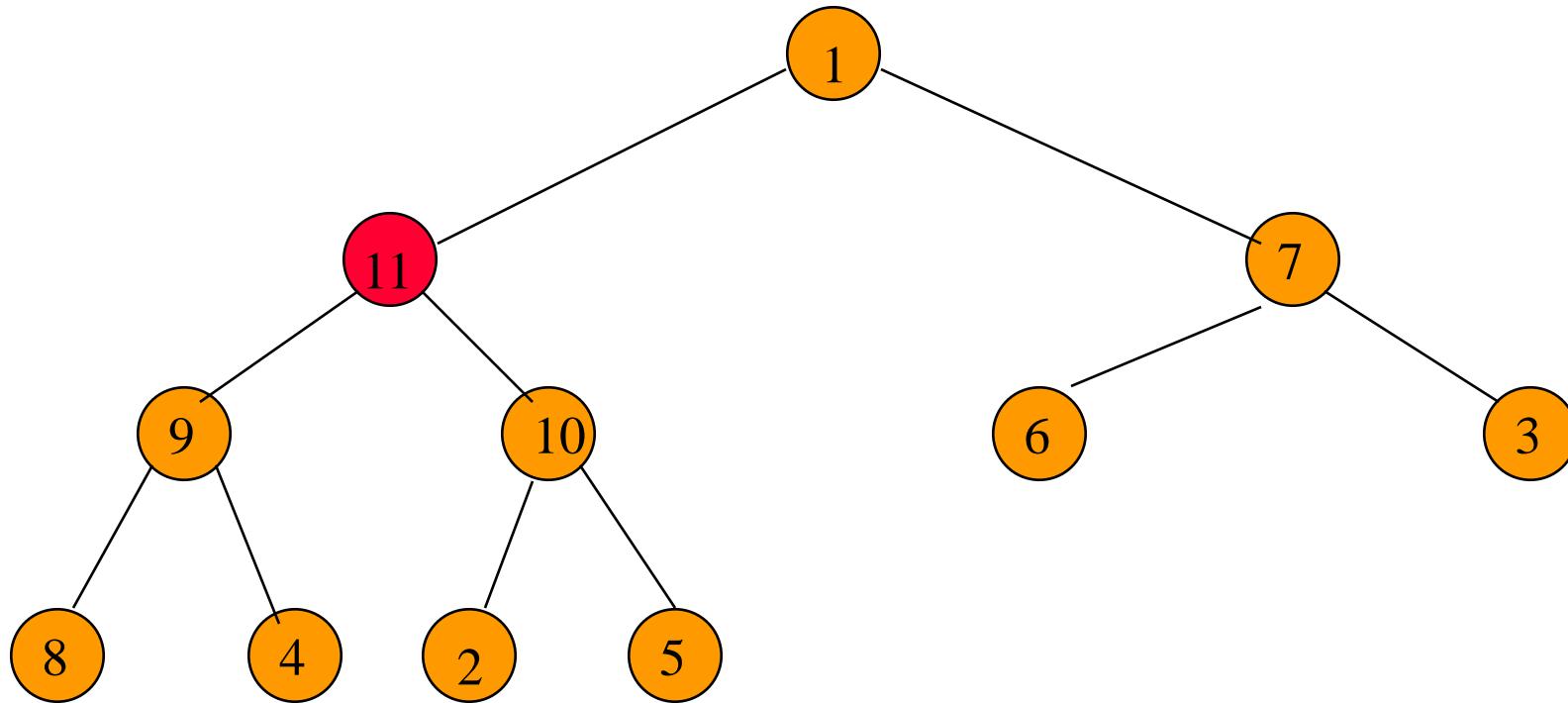
Initializing A Max Heap



input array = [1, 11, 7, 9, 10, 6, 3, 8, 4, 2, 5]

Find a home for 2.

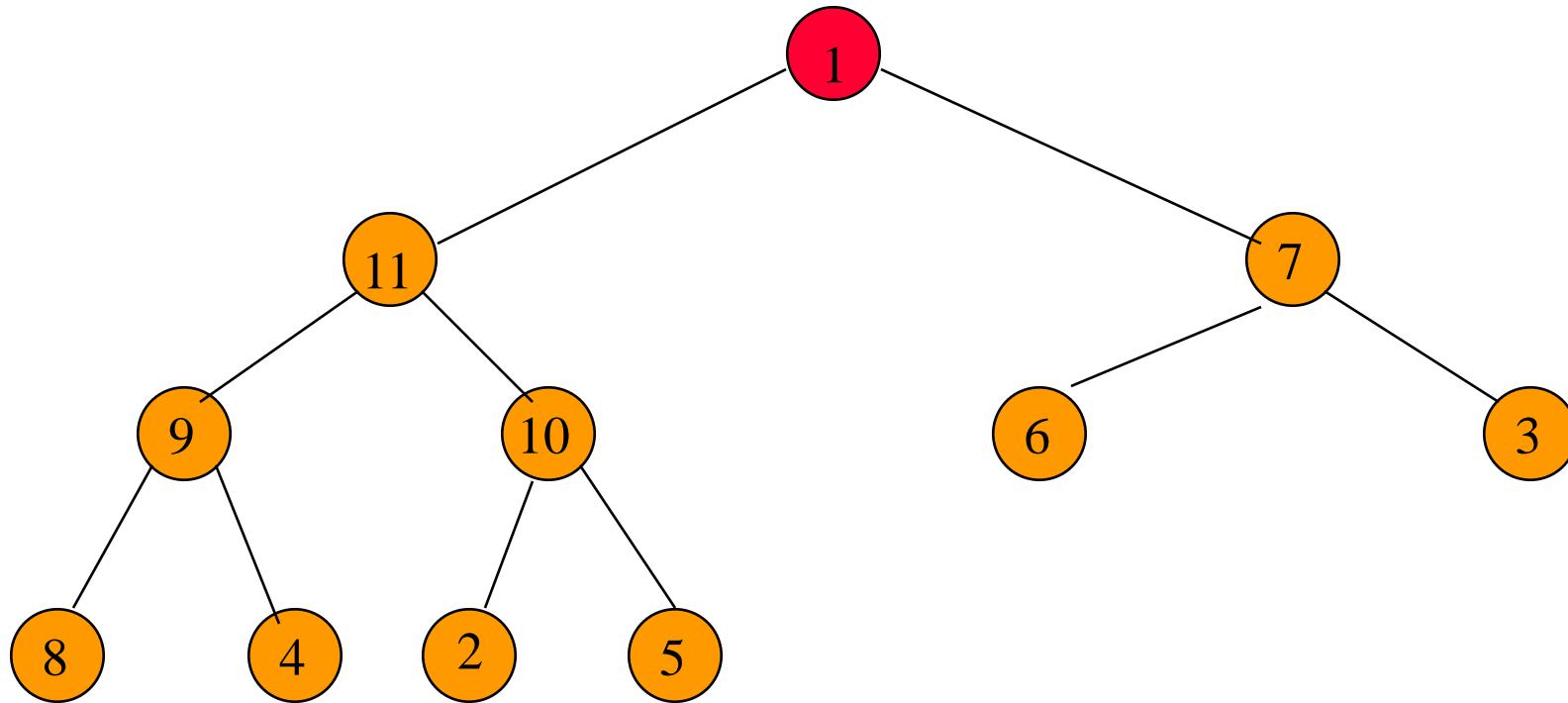
Initializing A Max Heap



input array = [1, 11, 7, 9, 10, 6, 3, 8, 4, 2, 5]

Done, move to next lower array position.

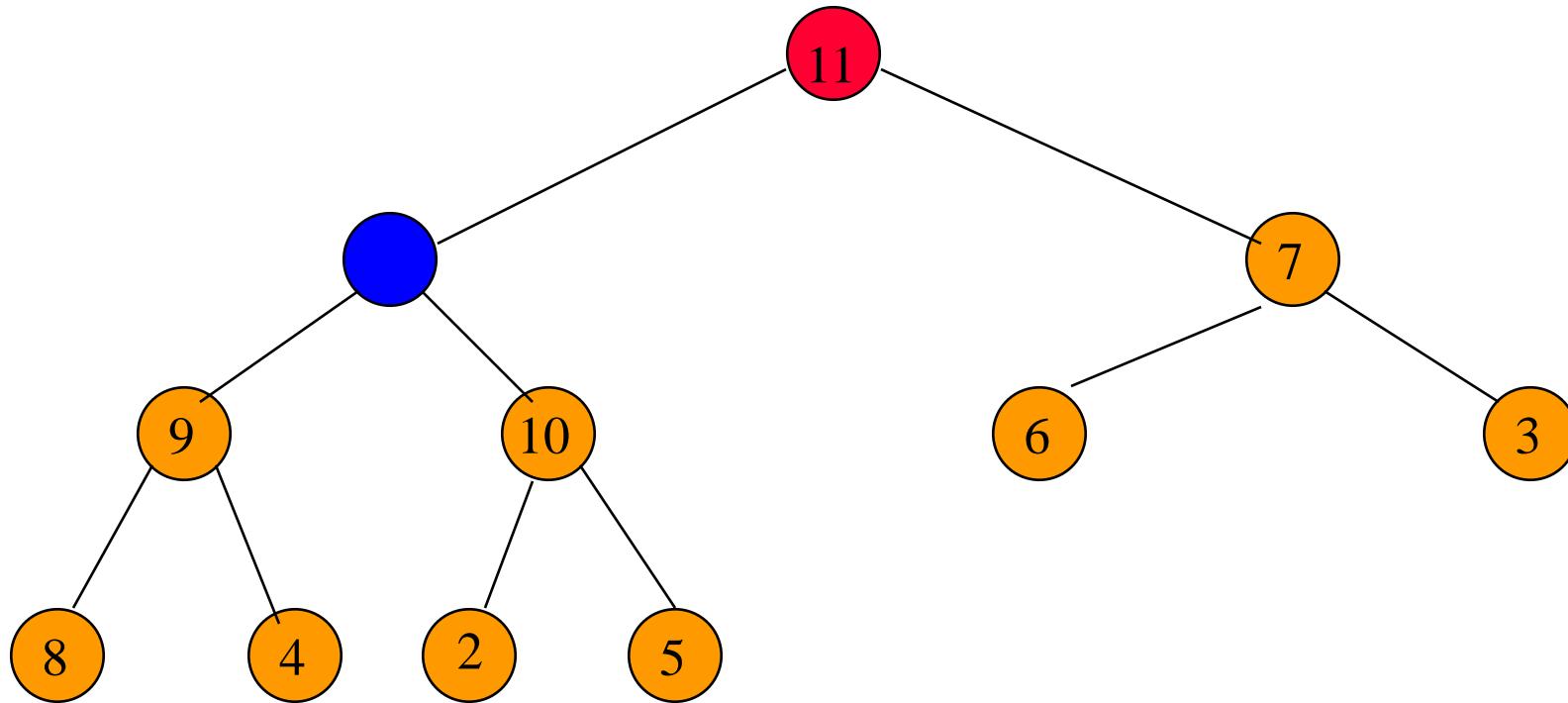
Initializing A Max Heap



input array = [1, 11, 7, 9, 10, 6, 3, 8, 4, 2, 5]

Find home for 1.

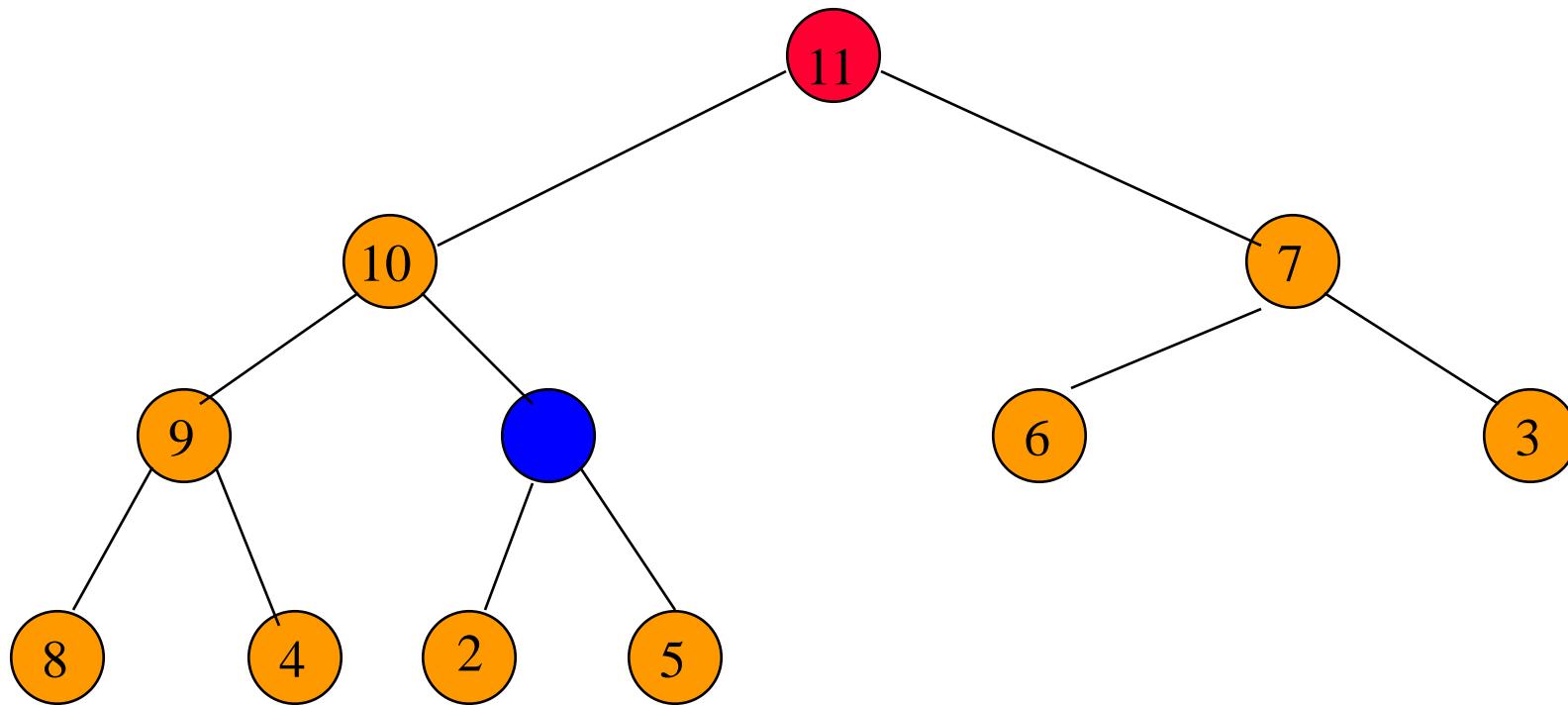
Initializing A Max Heap



input array = [11, 1, 7, 9, 10, 6, 3, 8, 4, 2, 5]

Find home for 1.

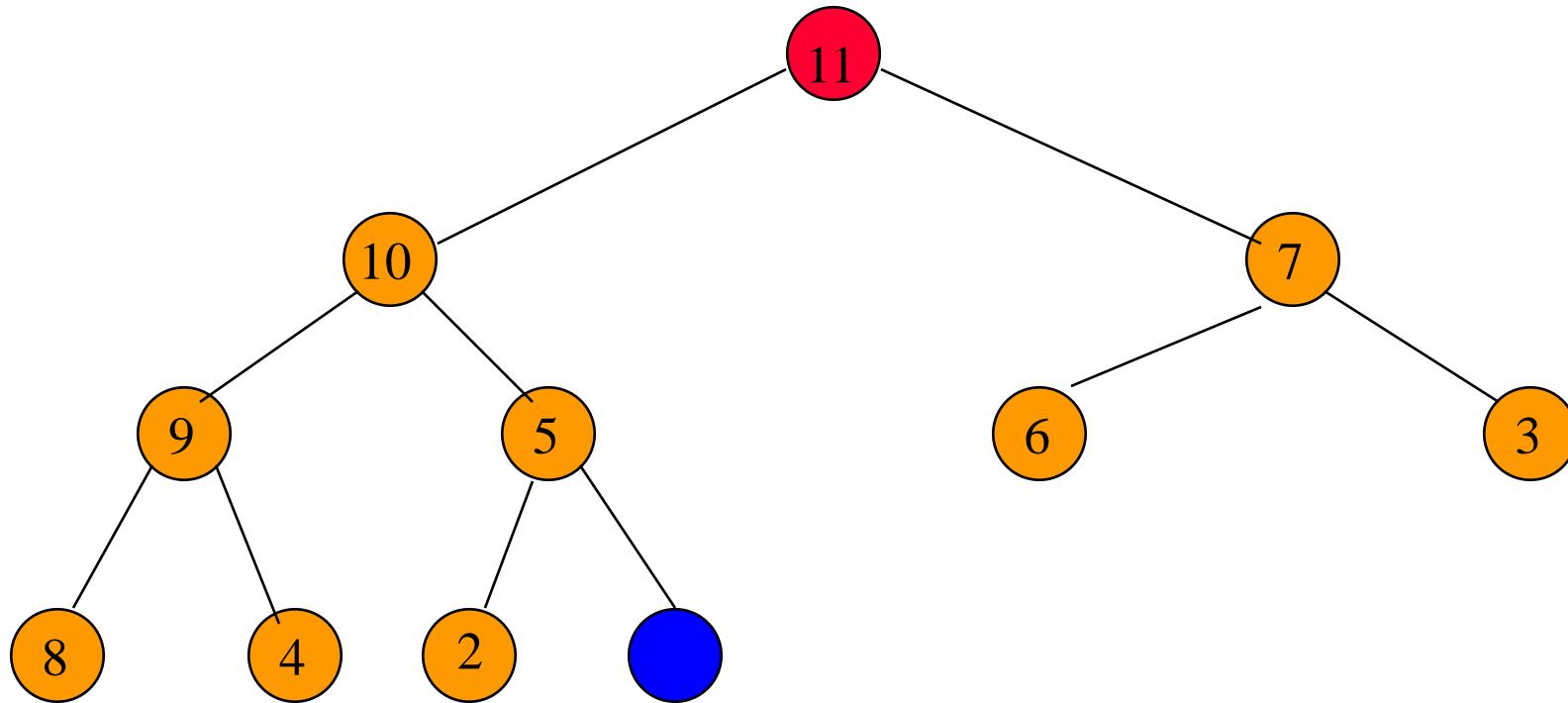
Initializing A Max Heap



input array = [11, 10, 7, 9, 1, 6, 3, 8, 4, 2, 5]

Find home for 1.

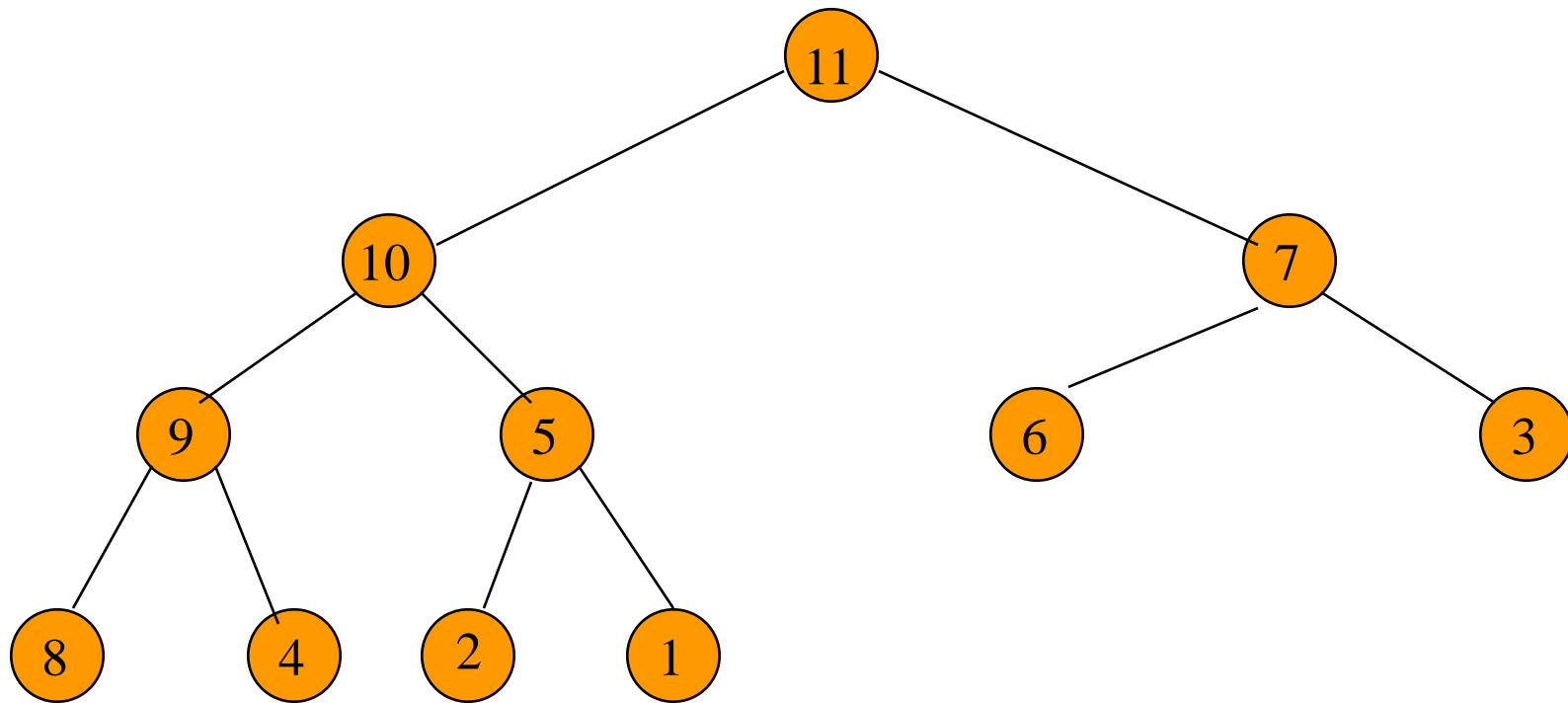
Initializing A Max Heap



input array = [11, 10, 7, 9, 5, 6, 3, 8, 4, 2, 1]

Find home for 1.

Initializing A Max Heap

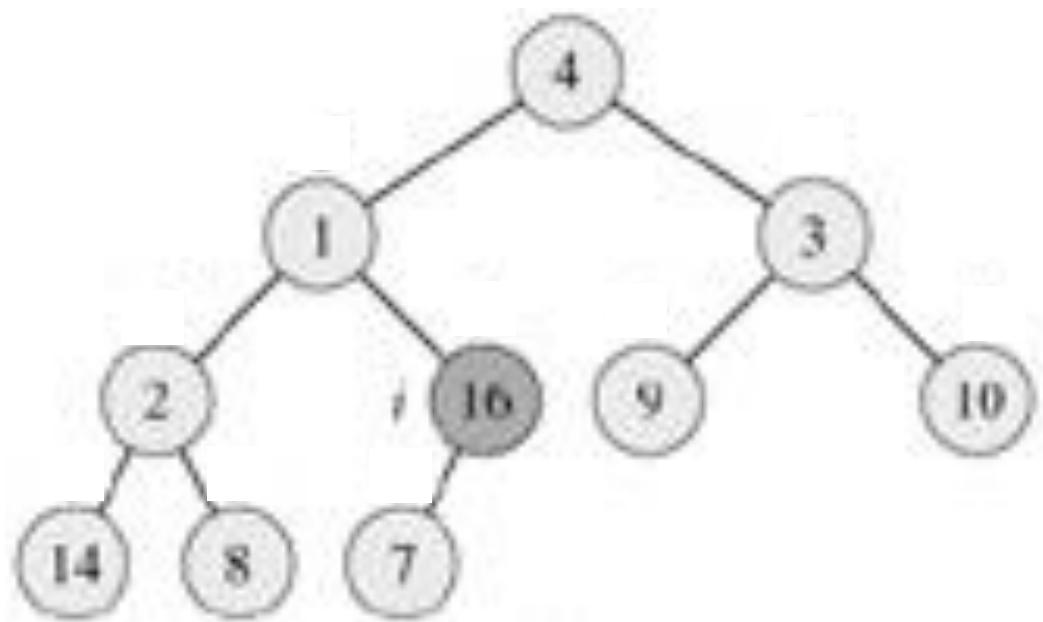


input array = [11, 10, 7, 9, 5, 6, 3, 8, 4, 2, 1]

Done.

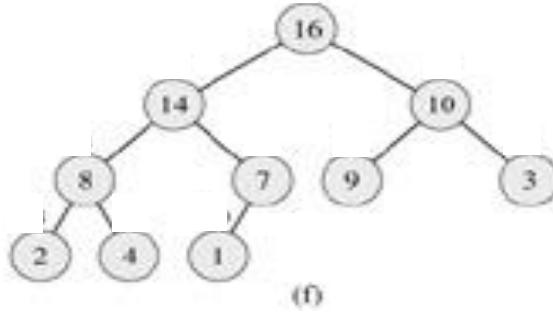
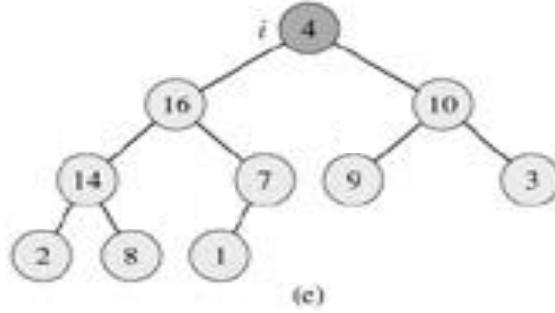
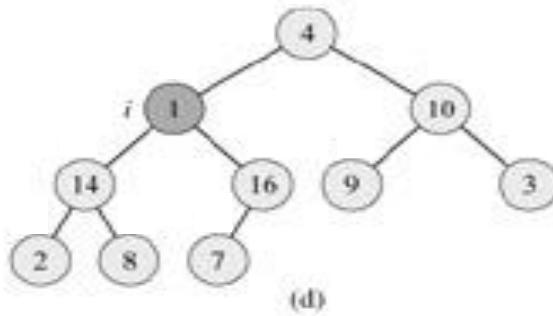
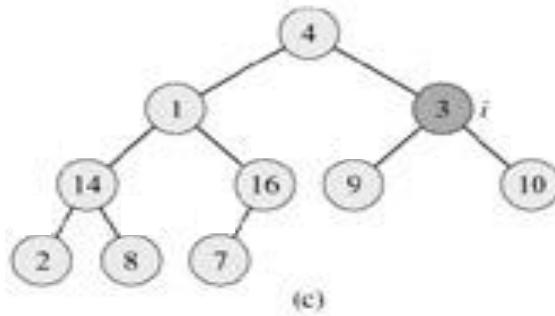
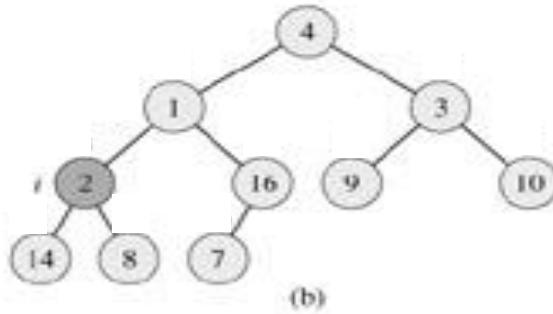
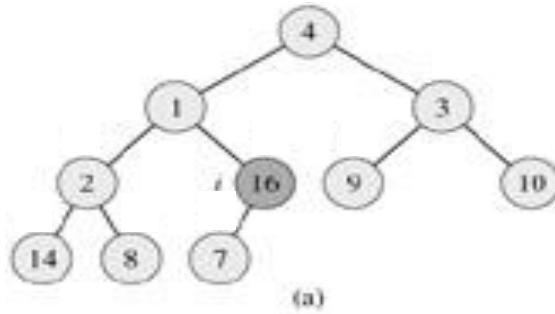
Initializing A Max Heap

A	4	1	3	2	16	9	10	14	8	7
---	---	---	---	---	----	---	----	----	---	---



Initializing A Max Heap

A [4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7]



MaxHeap initialize

```
#initialize max heap to element array Heap
def initialize(self, heap, size):
    self.size = size
    for i in range((self.size-2)/2, 0):
        self.maxHeapify(i)
```

Heap Sort

Uses a max priority queue that is implemented as a heap.

http://www.youtube.com/watch?v=WYII2Oau_VY



Heap Sort

```
# sort the elements a[1 : a.length - 1] using the
# heap sort method
def heapSort(a):
    # create a max heap of the elements
    h = MaxHeap(len(a))

    # extract one by one from the max heap
    for i in range(len(a) - 1, 0, -1):
        a[i] = h.removeMax(a, i+1)
```

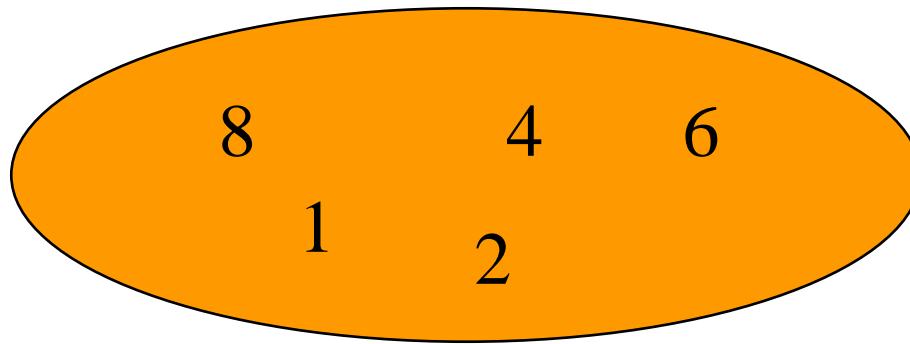
demo: HeapSort.py

Sorting Example

Sort five elements whose keys are 6, 8, 2, 4, 1 using a max priority queue.

- Put the five elements into a max priority queue.
- Do five remove max operations placing removed elements into the sorted array from right to left.

After Putting Into Max Priority Queue

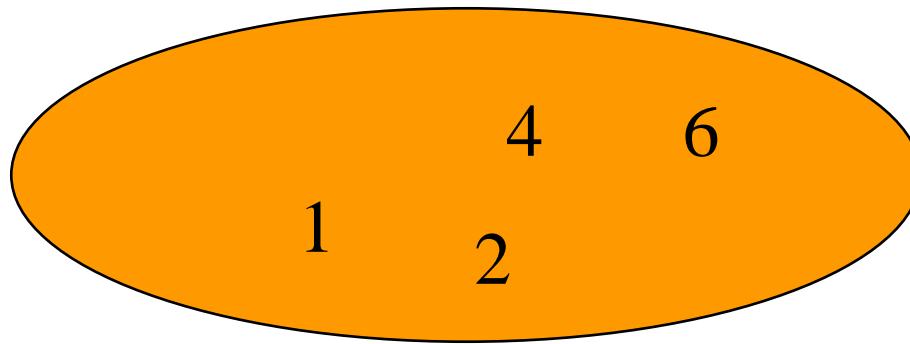


Max Priority
Queue

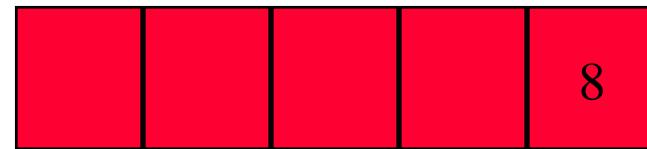


Sorted Array

After First Remove Max Operation

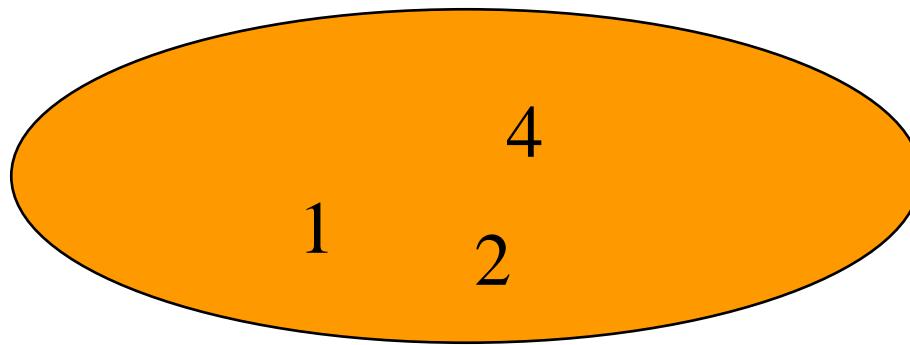


Max Priority
Queue

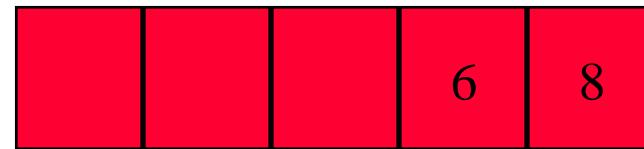


Sorted Array

After Second Remove Max Operation

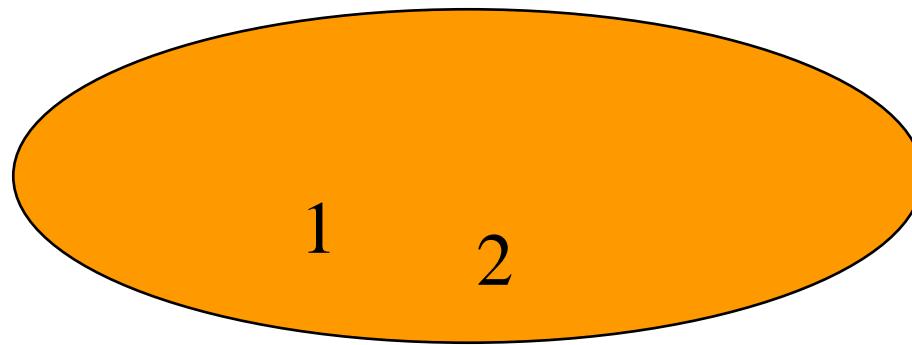


Max Priority
Queue



Sorted Array

After Third Remove Max Operation

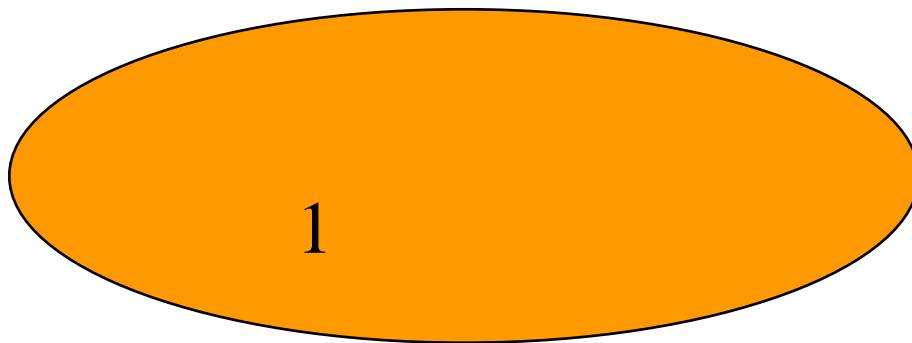


Max Priority
Queue



Sorted Array

After Fourth Remove Max Operation

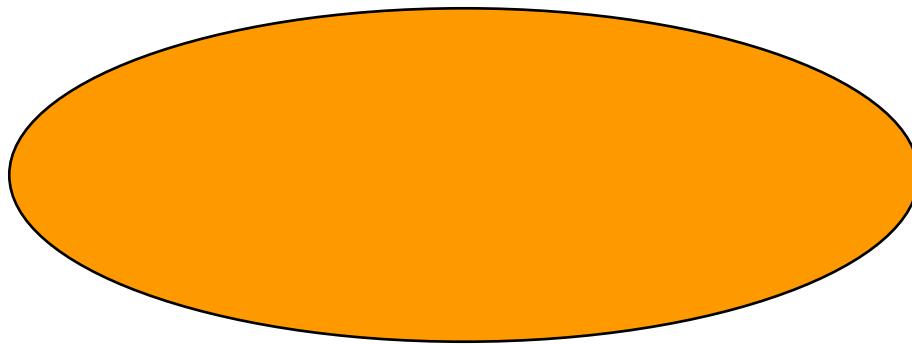


Max Priority
Queue



Sorted Array

After Fifth Remove Max Operation



Max Priority
Queue



Sorted Array